An Historical Based Adaptation Mechanism For BDI Agents

A thesis submitted for the degree of
Masters By Research

Toan Phung B.Sc.Comp Sci (Hons.),
School of Computer Science and Information Technology,
Science, Engineering, and Technology Portfolio,
RMIT University,
Melbourne, Victoria, Australia.

27th March, 2007

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Toan Phung
School of Computer Science and Information Technology
RMIT University
27th March, 2007

Acknowledgments

I would like to offer my insubstantial gratitude to Dr Michael Winikoff for his tireless efforts in helping to bring this thesis into fruition. I would also like to thank Professor Lin Padgham for her comments and feedback. A very warm thank you to Smart Internet CRC for their support and encouragement throughout my candidature.

Credits

Portions of the material in this thesis have previously appeared in the following publication:

- Learning Within The BDI Framework: An Empirical Analysis; Toan Phung, Michael Winikoff, Lin Padgham; Ninth International Conference on Knowledge-Based Intelligent Information & Engineering Systems (KES 05); September 2005 Melbourne, Australia.

This work was supported by the Smart Internet CRC.
All trademarks are the property of their respective owners.

Note

Unless otherwise stated, all fractional results have been rounded to the displayed number of decimal figures.

Contents

List of Figures

List of Tables

Summary

One of the limitations of the BDI (Belief-Desire-Intention) model is the lack of any explicit mechanisms within the architecture to be able to learn. In particular, BDI agents do not possess the ability to adapt based on past experience. This is important in dynamic environments as they can change, causing previously successful methods for achieving goals to become inefficient or ineffective. We present a model in which learning, analogous reasoning, data pruning and learner accuracy evaluation can be utilised by a BDI agent and verify this model experimentally using Inductive and Statistical learning.

Chapter 1

Introduction

Intelligent Agents are a new way of developing software applications. They are an amalgam of Artificial Intelligence (AI) and Software Engineering concepts that are highly suited to domains that are inherently complex and dynamic [Jennings, 2001; Wooldridge, 2002]. Agents are software entities that are autonomous, reactive, proactive, situated and social. They are *autonomous* in that they are able to make decisions on their own volition. They are *situated* in some environment and are *reactive* to this environment yet are also capable of *proactive* behaviour where they actively pursue goals. They are capable of *social* behaviour where communication can occur between agents. BDI (Belief Desire Intention) agents are one popular type of agent that support complex behaviour in dynamic environments [Bratman, 1987; Rao and Georgeff, 1995].

Agent adaptation can be viewed as the process of changing the way in which an agent achieves its goals. We distinguish between 'reactive' or *short-term adaptation*, 'long-term' or *historical adaptation* and 'very long term' or *evolutionary adaptation*. Short-term adaptation, an ability that current BDI agents already possess, involves reacting to changes in the environment and choosing alternative plans of action which may involve choosing new plans if the current plan fails. 'Long-term' or historical adaptation entails the use of past cases during the reasoning process which enables agents to avoid repeating past mistakes. 'Evolutionary adaptation' could involve the use of genetic programming or similar techniques to mutate plans to lead to altered behaviour. An example of short-term adaptation may be an agent that extinguishes fires. When a fire is encountered by the agent, it evaluates its plan-set and selects the most appropriate plan that extinguishes the current fire by covering the fire affected area with a retardant, say carbon dioxide. However, the agent would keep

no history of this encounter and would hence perform the exact same actions if a similar situation occurs again. For some situations that only require a reactive response where past instances are not needed, this may be enough. However, some problems require a more temporal approach where past experience needs to be considered. This is particularly important in dynamic environments that can change rapidly, causing methods for achieving goals that worked well previously to become inefficient or ineffective. Hence, 'long-term' adaptation needs to be introduced into the BDI framework. Typically, you would expect an intelligent entity to exhibit some form of improvement given more exposure to the same situation. However, this type of adaptive behaviour does not occur in current BDI systems [Georgeff et al., 1999].

Our work aims to improve BDI agents by introducing a framework that allows BDI agents to alter their behaviour based on past experience, i.e. to learn. An example of how learning agents can be useful can be seen in the work of Pereira and Costa [2000]. Here, learning agents are used to crawl and retrieve web pages which are then used as input to the agent's learning process. The web page data is categorised into two sections: (1) Link Learning data and (2) Keyword Learning data. Learnt link data is used to recommend similar pages to ones returned to the user whilst learnt keyword data is used to improve query searches by updating keywords in the agent's "keyword vector".

Learning from past experiences can be broken down into two distinct phases: (1) Storing past experiences and (2) Accessing these experiences for the purposes of improving performance. Hence, the following high-level questions are raised:

- How should an agent store its past experiences?
- How should an agent use these experiences to improve its performance?

With the question of storing past experiences, the issue of excessive data accumulation arises whereby storing every case encountered results in the gradual, sometimes dramatic increase of storage space required. When running such an agent learning system with finite storage space, this becomes an issue. Hence, investigating how to prune or filter such data seems appropriate. This would entail the creation of algorithms to manage and process historical cases as the agent's experience grows. Yet, at the same time this managing of past cases should not be detrimental to the performance of the agent, namely the accuracy of its predictions.

With regard to the second question, using past experiences presents a challenging set of problems: What learning algorithm should the agent use? When should it learn? Another

issue that arises is the question of reliability of such learnt knowledge. In other words, how can the agent be sure that the learnt information is accurate? If the learnt knowledge is known to be reliable, how exactly should the agent use that knowledge to improve its performance?

Hence, from the two questions above we propose to address the following more specific research questions:

- What type of learning should the agent use?
- How can an agent assess the accuracy/reliability of learning algorithm output?
- Can the use of analogous reasoning improve Statistical Learning?
- Does the placing of thresholds on Statistical Learning improve accuracy?
- What is the effect of pruning the history?
- When should an agent apply learning algorithms to stored history?
- What effect do domain characteristics have on learning?

The contributions of this thesis are:

- A model of learning that can be utilised by BDI agents
- An analogous reasoning algorithm, Simile, that can be used to improve learning accuracy
- A modification of traditional statistical learning to include clustering that improves learning accuracy
- A method for dynamically adjusting the trust an agent has of its learnt knowledge
- A quantitative comparison between Inductive and Statistical Learning
- Boosting of the predictive accuracy of Statistical learning through analogous reasoning
- The examination of the effects of historical pruning on Inductive and Statistical Learning

- An examination into the effects of the dynamic adjustment of acceptance of learnt data
- An investigation of how often an agent learns and its effects on their predictive accuracy
- An examination of domain characteristics and their effects on agent learning

To summarise the results of this thesis, the following tables outline comparisons between learning mechanisms, various algorithms and their effects on both Inductive and Statistical learning:

	Inductive Learning
Compared to Statistical Learning	More accurate but takes significantly longer
Analogous Reasoning	Not used
Historical Pruning	10% drop in accuracy
Dynamic trust adjustment	Made little to no difference
Frequency of learning	Accuracy of predictions is proportional to frequency of learning
Domain Characteristics	Exceptions to rules governing success have little effect

Figure 1.1: Summary of effects to Inductive learning

	Statistical Learning
Compared to Inductive Learning	Faster yet less accurate
Analogous Reasoning	Accuracy improved by 20%
Historical Pruning	3% drop in accuracy
Dynamic trust adjustment	Not used
Frequency of learning	Not used
Domain Characteristics	Exceptions to rules governing success has major effects

Figure 1.2: Summary of effects to Statistical learning

We are not developing new learning techniques per se, rather we propose a model for integrating learning into BDI agents and to experimentally validate that this model allows agents to improve their performance over time. The next chapter will cover the background necessary to understand the learning model described in Chapter 3. Chapter 2 also presents some related work within the context of the general field of agents and learning. Chapter 4 will present an analysis of our experimental results. Chapter 5 will describe additional related work in more detail while Chapter 6 will present our conclusions and future research avenues.

Chapter 2

Background

In this chapter we will introduce concepts that will be required knowledge for the understanding of future chapters. First we will introduce the concept of agents and the various research areas related to agent adaptation. Next we will describe a popular model of agency, the BDI (Belief-Desire-Intention) model of agency. As our work involves learning, we will give an introduction to the field of machine learning. Finally, we will introduce the domain in which we will conduct our research, *RoboCupRescue*.

2.1 Agents

Agents can be defined as autonomous entities that act within an environment, that is, agents are free to *choose* their own actions [Wooldridge, 2002]. Agents may *react* to stimulus from their environment and are also capable of *changing* their environment. In addition to these properties, *intelligent* agents are also *proactive* in that they are able to pursue goals and to remain committed to those goals and they are *social* in that they are able to communicate with other agents that may exist in the world. This social ability is not limited to the sending and receiving of messages but to also perform as a team where the use of social interaction plays a vital role in the coordination of a group of agents.

Situated in an environment, such an agent perceives its world through *sensors* and affects its world by a set of *actuators*. The process the agent employs to bring about an action begins with some stimulus from the environment which is perceived by its sensors. Information is extracted from the stimulus and is then fed into the reasoning engine. After some deliberation, the actuators are activated to perform an action. In goal driven agents, actions can also be initiated based on unfulfilled goals.

Figure 2.1: The Agent Concept

As seen in Figure 2.1 an agent is merely an entity that senses and acts. How it reasons and acts depends on the type of agent it is and what functionalities are available to the agent. The types of agents that exist in the literature include *reactive* agents, *mobile* agents and *deliberative* agents. The simplest of agents are *reactive* agents which do no reasoning in order to perform their tasks. Their behaviour can be easily mapped to a finite-state-machine where given a known input, a precise description of its actions can be determined. As their name suggests, *reactive* agents simply 'react' without 'thinking'. *Mobile* agents are agents which have the ability to move across networks. Physical agents such as robots may physically move and hence can be considered 'mobile' in the classical sense. However, software agents can also be mobile by being able to 'move' some or all of their program code. This allows a mobile agent to transfer its data and computational state from one machine host to another. Deliberative agents are more complicated in that they are able to 'think' and reason. Therefore deliberative agents are less predictable as their behaviour may change depending upon conditions and can therefore change. BDI agents and reactive agents belong to the set of deliberative agents.

Agents are considered by some as an extension of Object-Oriented Programming (OOP) [Odell, 2002]. The major difference between Object-Oriented and Agent-Oriented programming is the notion of *unit invocation*, that is, *when* can a function be called. With Object-Oriented systems, a *method* or *task must* be executed when it is called, that is the program has no 'choice' but to execute the function when ordered to. In an Agent-Oriented system the execution of such methods can be done *conditionally*. In other words, an agent has the choice of executing a method if it chooses to or is given a good reason, for example, an increase in its utility. Nothing outside of the agent can directly execute a method, the agent must choose to do so. This is what is known as *encapsulation of invocation* [Odell, 2002].

Another difference that separates the Agent-Oriented and Object-Oriented paradigms is

the concept of *goals*. With an Agent-Oriented system, failed tasks can be recovered from by selecting alternate actions while this is not explicitly catered for in Object-Oriented design. The use of goals helps in the task of selecting alternative actions as goals aid in 'means-end' reasoning.

The uses of agents range from simple monitoring tasks and interactive entertainment to complex reasoning and controlling tasks [Weyns et al., 2005; Hoen et al., 2005; Lee et al., 2005]. An example of the use of deliberative agents can be seen in [Belecheanu et al., 2006] where the JACK Intelligent Agents Toolkit is used to construct realistic military simulations of teams of agents. Human cognitive features such as fatigue and limited short-term memory are also modelled. These simulations are used for training purposes for the UK's Ministry of Defense. Another example of deliberative agents being used in industrial settings can be seen again in [Belecheanu et al., 2006] where agents are given the task of optimising a supply chain production problem whereby the production and distribution of liquid oxygen and liquid nitrogen at various power plants is determined based on demand and predicted energy usage forecasts. The agents dynamically adjust how much and where the liquid oxygen and liquid nitrogen are utilised and produced and hence automate the distribution of these materials. An example of the use of a mobile agent can be for the processing of large amounts of data that is distributed across a network [Loke, 2001]. Traditionally, software would have to download the data in order to access and process it. Yet, with mobile agents, the agent may move the parts of its code, which contain the functions that manipulate the data, to the location of the data itself. This process is known as *migration* and can reduce the amount of bandwidth which would have been taken up during the download of the data from the host machine to the agent's machine. The mobile agent would then move from one host machine to another, moving its code and the results of any calculations. At the conclusion of the mobile agent's task, the agent can either migrate back home to its original starting location or simply terminate itself at its current location.

2.1.1 The BDI Agent Architecture

The Belief-Desire-Intention (BDI) [Bratman, 1987] model is based on philosophical work by Bratman, which postulates that a rational entity must have three essential cognitive structures: (1) *Beliefs* (2) *Desires* and (3) *Intentions*. Beliefs are information about the world such as what day it is today or how much fuel is left in the car. Desires are outcomes that the agent wants to bring about. In a sense, they act as motivations for the agent's

actions. Intentions are courses of action that the agent has committed to. When going about executing an intention, the agent has the ability to stop executing and begin deliberation. This interleaving between execution and reasoning is a powerful feature of the BDI model. If during the process of trying to complete an intention, a change in the environment occurs that invalidates the intention's purpose, that is, the premise on which a task is undertaken is no longer valid, then that intention can be abandoned.

Bratman's notion of plans and intentions stemmed from the field of philosophy which highlighted the fact that humans have *goals* which they wished to achieve. In order to bring about these goals, humans form *plans*. These plans are sets of steps that are followed in order to achieve goals. By having goals, humans should be somewhat motivated to achieve some or all of them, hence they should be *proactive* in trying to achieve these goals. These ideas have been transferred into the computer science domain and have given rise to *BDI* software agents.

The work of Rao and Georgeff [1995] was an attempt at formalising the theoretical BDI model proposed by Bratman so that it could be viewed as a computational model that could be implemented and experimented with. Rao and Georgeff concluded that a computable and viable BDI implementation should possess the following three properties: (1) Be able to create and utilise plans that can be invoked based on the current environmental conditions, that is, be sensitive to the *context* of a stimulus (2) Have a balance between *reactive* and *goal-oriented* behaviour where the agent designer can easily specify when the agent should pursue goals and when to react to changes in the environment which may not necessarily be related to any current goals (3) Be able to describe and implement functionality at the abstract level rather than at a programming-language specific level. This would decrease development time significantly.

There currently exists a range of BDI platforms [Huber, 1999; d'Inverno et al., 1997; Georgeff and Lansky, 1987]. The platform we will be extending with learning will be the JACK Intelligent Agents Toolkit[1]. JACK (Java Agent Compiler and Kernel) is a commercial, industrial-strength Java-based intelligent agent platform used to implement BDI agents. Developed in Melbourne, Australia by *Agent-Oriented Software* (AOS), the toolkit is used in research, commercial and military settings. The main concepts that JACK extends Java with are: 'agents', 'events', 'plans' and 'beliefsets'. JACK is built using the Java programming language and allows the incorporation of Java code. The inbuilt execution engine and

[1] http://www.agent-oriented.com

accompanying algorithms form the core of the system. JACK also has messaging support to allow for agent communication. In general, JACK agents receive events and use their plans and beliefsets to reason and act on the world.

The way in which the computational BDI model presented by Rao and Georgeff works is that it begins with an initial 'event' that is followed through to a final set of 'actions'. Events can be generated *externally* by other sources or *internally* by the agent or other agents. The process flow begins with an event which is 'processed' by the agent. This agent possesses *goals*, a *plan library*, a set of *beliefs* and an *intention structure* [Bratman, 1987]. When an event is processed, a subset of plans from the plan library are selected based on how relevant they are to the incoming event. From this subset a further filter is applied to the context of the event to form an 'applicable plan-set'. One of these applicable plans is selected and placed onto the intention structure. The intention structure describes the agent's intended goals and by executing plans from this structure, external actions may occur, leading the agent to gradually achieve its goals. Figure 2.2 illustrates the BDI execution model.

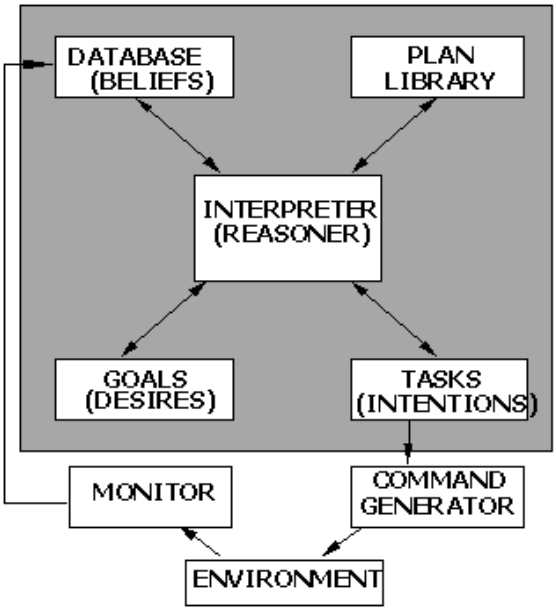

Figure 2.2: The BDI Execution Model (From [Georgeff and Lansky, 1987])

The source of an event can be either *internal* or *external*. Internal events are those that are created by the agent itself. This allows other *plans* that the agent may have to be invoked. External events are those which are not created by the agent and may be created by either

the environment or by other agents. With respect to the overall execution cycle of a BDI agent, an event may lead to some *goal* being achieved. When an event is perceived, the agent may choose to execute certain plans. Plans consist of *triggers*, *pre-conditions* and *plan bodies*. Plan triggers are what activate the plan and cause the agent to consider that plan. At this point, the pre-conditions need to be evaluated. A pre-condition is a proposition that, when evaluated, returns either 'true' or 'false'. Thus pre-conditions act as guard conditions. If 'true' is returned the plan is applicable. As an example, consider a plan that has the trigger 'Fire occurs', the pre-condition 'Is intense fire?' and a plan body of 'Extinguish fire'. When a fire occurs, the trigger is activated and the pre-condition is tested against the type of fire. If the fire is 'intense' then the pre-condition returns true and the plan body of 'Extinguish fire' is added to what is known as the *applicable plan set*. The applicable plan set contains all plans that are applicable in the current situation. Once all plans are evaluated, the applicable plan set is then consulted to provide a plan to execute, in this case the 'Extinguish fire' plan. However, there are short-comings with this model such as the lack of learning [Georgeff et al., 1999].

2.2 Machine Learning

Within a human context, learning can be defined as the process of improving one's performance over time through the acquisition of knowledge. Within a computational framework, "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [Mitchell, 1996]. Mitchell's definition describes the process of improving performance over time within a set of given tasks.

Learning can be very useful in domains with high levels of unpredictability. This is because the designer of a software solution to a problem that possesses high levels of unpredictability can not possibly think of every eventuality that can occur, hence writing software to deal with these situations is very difficult. If the system is provided with a means by which experience can be used to improve performance, then unexpected situations can be handled more effectively. This, however comes at a cost. The process of learning can be very expensive and therefore may impede on the overall performance of the system the learning algorithm is being applied to. Hence the benefits of learning in a system must always be weighed against its cost.

There are many different types of learning techniques available with each type of learning

having its own benefits and disadvantages. Techniques such as *case-based reasoning*, *neural networks*, *Bayesian learning*, *reinforcement learning*, *inductive logic programming*, and *genetic algorithms* can be categorised as machine learning techniques.

A popular sub-branch of machine learning, *data mining* is an area of research whereby patterns are analysed and extracted using machine learning techniques. This information is most widely used for such purposes as marketing or sales whereby the shopping patterns of customers can be used to direct advertisements and product placements that target certain clientele. The three main types of data mining techniques that exist are *classification*, *clustering* and *association rule finding* [Han and Kamber, 2001]. Classification takes a single attribute or value and attempts to categorise it into two or more 'buckets' or categories. For example, when given a description of a mushroom the classifier must determine whether it is safe to eat or is poisonous. Clustering is similar to classification except that instead of evaluating single attributes against a rule it takes multiple attributes or values and evaluates them against each other. Hence, the number of buckets is not known until the end when a result is given. The result produced is a set of 'clusters' or groups of attributes where each group member possesses some common trait(s). Association rule finding is similar to clustering except that a set of rules that emphasise key attributes is created rather than a set of clusters.

The work of Symeonidis et al. [2002]; Mitkas et al. [2003] makes an attempt at integrating data mining techniques into the BDI framework. In particular they use classification, clustering and association rule algorithms to discover patterns and rules that may be useful to BDI agent reasoning. They describe how agents can be created in 3 primary ways: (i) with enough intelligence from the beginning (ii) be pre-trained before being activated or (iii) with no prior knowledge of its tasks which have to be learnt by the agent itself. Their argument is that although agents can be designed with the needed intelligence, there may be exceptional circumstances that the designer did not think of and as a result, the agent would not be able to adapt to those changes. The solution proposed by Symeonidis et al is the *Agent Academy* platform that is capable of re-training agents should they require any new knowledge. The source of this knowledge initially comes from the Web which is retrieved to form a base knowledge set. The technology used to realise the Agent Academy platform consists of many well known systems. These are JADE (to implement the agents), JESS (rule-based reasoning), WEKA (data mining), ZEUS (for agent mobility) and Protege-2000 (ontologies). The specific types of data mining algorithms used are ID3, C4.5 and $\theta - FLNMAP$ which are all decision tree algorithms. K-means is used as the clustering algorithm.

The work done by Tambe et al. [2000] describes the *TeamCore* framework in which multiple agents use machine learning to cooperate with each other. Their work focusses on the notion of 'team adaptation' with four main components: (1) Adaptive autonomy (2) Adaptive Execution (3) Adaptive monitoring and (4) Adaptive Information Delivery. The Adaptive Autonomy component uses the C4.5 learning algorithm with user feedback and hence is a supervised learner. Adaptive Execution uses probabilistic reasoning in addition with utility functions to choose actions. Adaptive monitoring allows the agents to infer progress from coordination messages. The agents are even able to predict messages from other agents and to give feedback to messages that may in turn change the behaviour of other agents. The Adaptive Information Delivery component provides a utility by which the value of message is weighed against the cost of reinforcement learning, as learning can be an expensive process, this mechanism allows for potential saving of resources.

2.2.1 Reinforcement Learning

Reinforcement learning is a type of learning that *rewards actions that lead to positive outcomes*. As the name suggests, positive actions are *reinforced* while negative actions are not. Over time, this leads to a system that is trained to perform actions that lead to successful outcomes. An analogy of this type of learning can be seen in the training of animals, such as dogs. When the dog performs a 'positive' action, such as fetching a stick, it is usually rewarded with praise and a treat. Over time, the dog learns that some actions are more favourable than others and hence its behaviour is veered towards some actions more than others.

The most popular type of reinforcement learning by far in the literature is *Q-Learning* where 'Q' stands for 'quick'. Q-Learning uses a function to estimate the potential reward for a particular action a given a state s. This is known as the *Q-Function* and is represented as $Q(s,a)$ where 's' and 'a' are the state and action respectively. Q-Learning is favoured in the literature because it allows agents to learn the mapping of states to actions without any explicit model of the environment. As with reinforcement learning, the data which describes positive and negative reinforcement toward an action is stored within a table, known as a *Q-Table*. However, a short-coming of Q-learning is that the tables that keep the reinforcements tends to grow quickly, hence the need to manage the size of the table is important in large and dynamic domains [Bruske et al., 1996].

A recent advancement in reinforcement learning that is relevant to the work presented

in this thesis was presented by Dzeroski et al. [1998] in which an interesting approach using logic, may prove more efficient than traditional Machine Learning algorithms. Essentially their work uses reinforcement learning (Q-learning) combined with logical relational operators that allow a richer set of objects to be described. State representation is given as a triple where the *state*, *goal* and *action* are combined to give a Q-value.

2.2.2 Inductive Logic Programming

Inductive Logic Programming (ILP) is a method of learning that uses the process of induction [Muggleton, 1992] to produce a set of rules with a certain predictive power. Induction is the process by which given (1) A set of positive examples (2) A set of negative examples (3) Background knowledge and (4) A hypothesis language, a set of rules is created that describes all of the positive examples and none of the negative examples. For example, given a set of positive examples that consisted of 'Used the green formula' and a set of negative examples that consisted of 'Used the red formula', we would expect an inductive learner to produce a rule like '*Use the green formula otherwise it will fail*'.

Within our fire fighting domain, the background knowledge consists of various attributes of fires such as *intensity*, the *size of the building*, the amount of *combustible material* and the *weather conditions*. In addition to this, the relationship between all those attributes and how they combine to form the definition of a single fire are provided in the background knowledge. The hypothesis language is what is used to describe or represent the output of the learning process i.e a set of rules. Without a proper hypothesis language i.e one that has enough expressive power, the predictive power of the induced rules may be severely limited.

As an example, consider the following input:

```
Positive Examples
-----------------
<Mild, Wooden, Foam, Fire Extinguished>
<Windy, Steel, Halon, Fire Extinguished>
<Overcast, Concrete, Halon, Fire Extinguished>

Negative Examples
-----------------
<Windy, Wooden, Water, Fire Burning>
<Overcast, Concrete, Carbon Dioxide, Fire Burning>
```

```
<Mild, Steel, Dry Chemical, Fire Burning>
```

The tuples above represent <Weather, Building Type, Extinguisher, Outcome>. As seen, there exist two sets of examples, positive and negative. When these examples are supplied to an inductive logic program, the output is a set of rules that describes all of the positive examples while not representing any of the negative examples. For example, the rule: *If Halon or Foam are used, the fire will be successfully extinguished.*. In terms of computability, a more appropriate way of expressing the previous rule would be:

```
IF Retardant = Foam || Retardant = Halon THEN
    Fire Extinguished
ELSE
    Fire Burning
```

As seen in figure 2.3, the tree has a root node which is followed by a series of conditional paths connected to other nodes. These paths are followed until a terminal node is reached at which a solution is given. Another popular method of expressing rules is in the form of a *decision tree*.

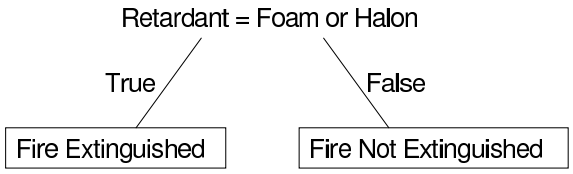

Figure 2.3: A Decision Tree

An example of inductive learning can be seen in the work of Alonso and Kudenko [1999]. Their work involves the combination of Inductive Logic Programming and Explanation-Based Learning(EBL). Explanation-Based Learning is an adaptation technique in which computation is saved by not calculating the same solution over and over. Instead, if the same problem arises a previously calculated solution is provided. For example, if a calculation involving the exact same numbers is encountered again, then there is no need to calculate it again, just use the answer you previously gave. This type of learning is also known as *speed up* learning. Hence, EBL is used to improve the *efficiency* of a problem solver whereas ILP is used to *acquire new knowledge*. This work is very interesting as they deviate from traditional machine learning, such as neural networks, which do not scale very well.

The introduction of inductive logic programming into agents has been explored by Matsui et al. [2000]. Their approach involved the use of a planner and learner to achieve adaptation. The system proposed in [Matsui et al., 2000] consists of five components: (1) An Observer, which converts the perceived world into first order logic clauses (2) A planner, which creates plans described in first order logic (3) An actor, which executes plans to perform actions (4) A learner, which uses inductive logic programming to create predictive rules and (5) The Checker, which decides what actions to perform based on the rules generated by the learner. The scope of their work entailed a soccer goal-kicking domain where the learning task was to predict where the agent should kick the ball from in order to score a goal. The training examples used for the learner consisted of 221 cases with 80 being successful goals and the remaining 141 tries being failures. Their results were very encouraging with their learning agent improving its performance from 36.2% without learning to 90.3% with learning with a standard deviation of 2.4%.

Alkemy

Alkemy [Ng, 2004] is a symbolic inductive learner written in C++. It uses Inductive Logic Programming to produce a decision tree. An example decision tree that Alkemy produces can be seen in Figure 2.4. Each node in the decision tree generated by Alkemy contains a higher order function that takes an *Individual* and returns a Boolean. Higher order functions are functions that can take other functions as arguments. The advantage this has over typical decision trees is that these higher order functions are capable of expressing more complex expressions such as relationships between objects rather than non-functional atomic values such as integers.

Alkemy takes the following parameters as input:

1. The domain, in terms of data types;
2. The function being learned;
3. A collection of training examples;
4. A hypothesis language, defined in terms of higher order functions; and
5. A rewrite system defining the search space for hypotheses.

The domain definition expresses the domain's data types in terms of the data types provided by Alkemy, such as integers, strings, boolean, trees and so on. For example, in our

domain we define the *Weather* to be either *Hot*, *Mild*, *Windy*, or *Rain*, and similarly for the other attributes of a fire. These domain definitions form the basis for defining the function that Alkemy is to induce from the training examples. The training examples are the past instances that have been stored by the agent.

The *hypothesis language* is defined by specifying a collection of higher order functions that can be used in the hypothesis. These functions represent the background knowledge, that is, how an object in the world can be evaluated. Each node in the decision tree generated by Alkemy contains a higher order function that takes an *Individual* and returns a boolean.

Alkemy operates by searching a space of possible hypotheses. The search space is defined using a set of rewrite rules. This gives a high degree of control over Alkemy's search process. The predicate rewrite system describes the hypothesis search space and consists of a set of rules that describe what a predicate can and cannot be. In a sense, the search through the hypothesis space is guided by the rewrite rules. These rewrite rules allow for the use of domain knowledge which is then used to direct the search.

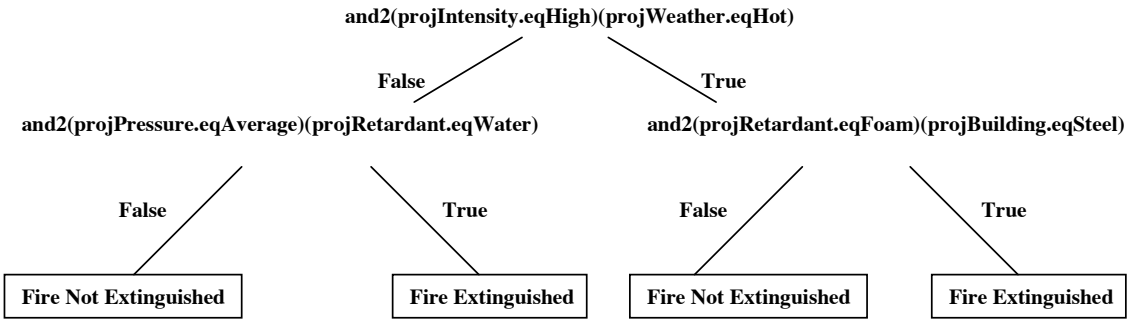

Figure 2.4: Higher Order Function Tree

As seen in Figure 2.4, the part of the root node that specifies *projIntensity . eqHigh* is the function that takes the Individual 'Intensity' and returns true iff its Intensity is High. The whole root node expression denotes a function that takes an Individual and returns true iff its Intensity is 'High' *and* its Weather is 'Hot'.

2.2.3 Case Based Reasoning

Case Based Reasoning is a learning technique whereby past cases are stored and retrieved at a later time for analogical purposes. When a situation occurs, the most relevant past cases are retrieved and analysed to see whether the solution that solved that past problem can be

reused to solve the current problem. This is much like the way in which people draw upon past experience to solve a problem.

Specifically, the process of case based reasoning consists of the following steps [Russell and Norvig, 2003]:

- Retrieve the most relevant past cases from the set of past cases
- Use those cases to solve the current problem
- Alter the past solution if it is not directly applicable to the current problem
- Insert the solution to the current problem as a new case into the set of past cases

The Case-Based BDI system in Olivia et al. [1999] uses a *concept hierarchy* to find information on the WWW if no similar cases are found while we assume no additional information sources and hence use *Simile* to reason further on existing information. The notion of 'easier' and 'harder' for case similarity is absent in [Olivia et al., 1999] however their model applies case reasoning on agent beliefs while we do not.

2.2.4 Psychological Approaches to Agent Learning

Since intelligent agents are described in terms of mental attitudes, it is not surprising that psychology has also been used to inspire agent learning mechanisms. Since the early work of people such as Georgeff and Lansky [1987] and Cohen and Levesque [1990], much work has been done on extending agents with more human-like properties such as learning and emotions. The work done by Zhang and Covaci [2002] focussed on adaptive personalised agents that possess emotions. Their system learnt through means of neural networks with the knowledge representation being implemented with a semantic network where objects with related features were associated with one another. The work by Norling [2001] extended the BDI model of agency to incorporate a more human-like reasoning model, namely Naturalistic Decision Making (NDM) theory and Recognition-primed decision making (RPD). The purpose was to allow for real-time adjustment in highly dynamic domains such as the *Quake 2* 3D computer game. Norling's work allowed agents to achieve adaptability via *cues* that describe subtle differences between similar scenarios. These cues reduce the search space and provide abstraction. Much like our work, the designer of the agent system provides domain specific knowledge.

2.2.5 Biological Approaches to Agent Learning

The process of adaptation and learning can be seen abundantly in nature with animal and plant species altering anything from their behaviour, environment or genetic makeup/mutation. This has given rise to a biologically inspired branch of agent research in which various techniques that appear in nature are incorporated into agents to facilitate adaptation and learning. Such techniques include the use Genetic Programming (GP), Neural Networks (NN) and even the use of artificial pheromone marking as seen in Sauter et al. [2002].

Neural Networks (NN) are a set of interconnected nodes where input is taken in one end and output is produced on the opposite end [Russell and Norvig, 2003]. This structure is inspired by the way neural cells are connected in the human brain. Typically, there are many layers of nodes where each layer is connected to the previous layer. Data is transmitted to each layer for processing and calculation. The two main data processing concepts in NN's are *thresholds* and *biases*. Thresholds are numeric values that are applied to the connections that exist between nodes. Biases are numeric values that are applied to the nodes. Together, thresholds and biases act as a 'guard' condition which dictates whether a node should 'fire' and pass down data to stimulate the nodes it is connected to. The calculation to determine whether the next layer of nodes should be activated is a simple multiplication of all the current thresholds against the input received. These are then summed. If the result is greater than or equal to the bias of the node it is connected to, then the next node will fire and process continues until either the end node is reached or until all stimulus has been removed by the biases. The process of learning involves the adjustment of thresholds and biases. In a mature neural network, the data may be skewed towards one particular path instead of the many others. This is because a type of conditioning or positive reinforcement has been applied to this path, making it more desirable to select this path. Similarly, negative reinforcements are applied to less desirable paths, making those actions less likely to occur. The advantages of Neural Networks are that they can handle noisy data very well. Some disadvantages include the issue of *over fitting* whereby the data entering the neural network is taken too literally and hence the network is tailored specifically to that data. This results from a small input data set so when new data from the same domain is applied to the Neural Network, it incorrectly classifies the data. Bayesian learning is similar to Neural Networks except that uncertain information is handled by applying probabilities to incoming data. Bayesian learners do not suffer from over-fitting like Neural Networks however the probabilities of the data must be known beforehand.

Genetic Programming (GP) is a genetically inspired method of evolving computational behaviour [Russell and Norvig, 2003]. It involves the creation of a random *population* whereby each individual possesses a certain *fitness level*. The fitness level is defined as percentage or probability of yielding a desired result. These individuals then use the process of natural selection to produce offspring that possess the traits of their parents. Computationally, this entails that two 'parent' programs merge their program code to produce a new 'child' program. This is what is known as one *cycle* which produces a new *generation*. The process of producing new generations is continued until either the fitness of the population does not improve or until x number of generations is produced. The advantages of Genetic Programming are that not much background knowledge is required and that they are relatively easy to implement. The disadvantages of Genetic Programming are that significantly large populations are required. This *genetic diversity* is needed to cater for as many relevant possibilities as required the problem. Also, the time needed to produce a reasonable solution may be quite large.

An example of work involving the use of genetic algorithms can be seen in the work of Smith and Taylor [1998]. Here, Smith presents a framework that allows the testing of genetic algorithms and other forms of evolutionary computation in agents. The agents perform cross-over of genetic encodings represented as character strings to produce values which are then evaluated using their framework. The performance of the cross-over technique is then returned by the framework. One such approach where an agent uses genetic algorithms to evolve solutions based on the current situation can be seen in [Singleton, 2002]. Here, Singleton uses genetic algorithms to solve the Maes Action Selection Problem where the best action to take at that time is evolved to produce better solutions. As a result, the agent adapts through experience.

The work of Meyer [1997] uses agents to encapsulate genetic algorithms to help the agents (Animats) adapt to changes. Animats, short for artificial animals, are agents that simulate animals. They have a set of basic needs and desires such as the desire for food and mating partners. Meyer states that animal morphology has established that a bulb-like structure at the base of the brain, known as the 'Hippocampus' in animals is responsible for path planning. However, the computational realisation of this requires a by-pass of such a structure due to the complexity of such a construct. This is where genetic algorithms are applied in order to allow the animats to evolve plans for path finding.

Iba and Takefuji [1998] presents an adaptive learning agent framework which uses a combination of neural networks and genetic algorithms. Their focus is upon how nature can

be defined as evolution plus learning. Their view of evolution was that it is a 'population level adaptation' while learning is an 'individual level adaptation'. The genetic algorithms provide the population level and create new populations which are hybrids of the previous population while the individual level adaptation is achieved via a neural network.

Some agents such as BDI agents have *plans* which when executed, achieve some task. By switching plans based on a specific scenario an agent can adapt its behaviour to that scenario. Smith et al. [1999] present an approach that alters plans through Evolutionary Computing (EC). This is similar to the research domain of planning, except that biological adaptation is the inspiration. Their view is that since EC is a "naturally distributed AI technique", the cross-over of multiple individuals leads to emergent adaptive behaviour.

The work presented in [de Medeiros Rivero et al., 1998] uses a combination of genetic algorithms, *causal* systems and *anticipitory* systems to achieve adaptability. Causal systems are systems completely determined by the past. Anticipatory systems monitor environmental states and activate certain actions in response. Actions are encoded into genetic strings which are crossed-over to produce hybrid actions. These actions are then executed and a relevant fitness function is assigned to the hybrid action. The causal and anticipatory systems combined provide a means by which the genetic population can be tested for fitness.

2.2.6 Anthropological Approaches to Agent Adaptation

Adaptation in a multiagent context can be facilitated by the exchange of beliefs and knowledge from two or more agents that may be residing in more than one environment. The notions of agent societies and agent cities[2] have arisen that describe locales of heterogeneous agents that are distributed across large distances, usually across countries. With each set of agents belonging to a distinct group of users and developers, the issue of standards in agent languages and protocols becomes increasingly important. Being able to adapt to such *cultural* changes seems both a challenging and interesting problem. One novel approach to this problem states that agent adaptation in this context can be viewed from an anthropological perspective [Bordini and Campbell, 1995].

What Bordini and Campbell [1995] propose is the use of a subset of the broad field of Anthropology (Cognitive Anthropology) to allow an agent to create a model of its new social environment complete with social rules and norms and to use this model to help guide the agent's actions. Their approach can be explained through the metaphor of human migration.

[2] www.agentcities.org

When an individual or group of individuals moves from one society to another, they will need to learn the rules of the society they are migrating into. A particular action that is socially acceptable in a previous society may not be acceptable in the new society. The ways in which objects and the relationships between such objects are viewed differently by different groups of agents forms the basis for their approach to adaptation in a multiagent context in that it is a process of discovering what values and views the new society holds.

Specifically, Bordini and Campbell use such methods as *Adoption of Language*, *Participant Observation*, *Informants*, *Field Notes* and *Arrangement of Semantic Domains*. The adoption of language or some subset of the language used in the new society is important as it allows the agent to communicate with other agents who may be able to provide useful information or services. In addition to this, their definition of language also extends from verbal/sybolic semantics to gestures which may be "universally understandable". Participant Observation is one method that can be employed by a foreign agent to gain insights into the rules and norms of the society the agent is in. Specifically, the foreign agent should "...take part in the general 'problem solving' that is occuring in that society.". During this process, the agent is exposed to what actions and protocols are expected of it, which if considered by the agent in future acts of reasoning, would lead to adaptive behaviour. Informants within the newly migrated society can also be useful in helping the migrant agent to understand its new environment. Informants are other agents that can aid in adaptation by actions such as sharing protocols, helping define ontological terms or by clarifying language difficulties the migrant agent might have. Informants can be one of three types, namely *compatriot agents*, *sympathetic agents* and *novice master agents*. Compatriot agents are agents that have come from the same previous society as the migrant agent and so may have experienced similar problems. As a result, these compatriot agents would be helpful in solving some of the problems the migrant agent may have. Sympathetic agents are native agents that are knowledgeable about the society and are willing to help. The main difference between compatriot agents and sympathetic agents lies within their origins, that is, whether they are native to the society or have been previous migrants themselves. Novice-Master agents are agents that are part of the society who have the explicit task of "taking-in" new migrant agents and "showing them the ropes". This may not differ much from the previous two kinds of informants, but has a more formal relationship being formed between the participants. One of the assumptions made by Bordini and Campbell are that some of the agents in the new society are *altruistic*, that is they are willing to help without any consideration for reward or benefit.

Field notes and the arrangement of semantic domains both relate to the way in which knowledge is represented and arranged within the agent. Field notes are of the form

```
#<fieldnote_number> <date> <place> <name_of_informant> <topical_codes> <content>
```

where $\langle fieldnote\_number \rangle$ is a unique ID for reference purposes, $\langle date \rangle$ is the calendar date, $\langle place \rangle$ is physical location the observation took place, $\langle name\_of\_informant \rangle$ is the name of the informant, $\langle topical\_codes \rangle$ are "...(either personal or standard) codes for the topics covered" and $\langle content \rangle$ being the actual content of the observation. Using the formalisation of field notes and semantic domains, Bordini and Campbell present two main classes of agents: *Anthropologist* agents and *Migratable* agents. Anthropologist agents are agents that have the explicit goal of migrating from society to society with the aim of creating descriptions of those societies. These descriptions can then be exploited by newly arriving 'migratable' agents.

The work of Carabelea [2001] uses 'trust' and 'promises' to facilitate adaptive behaviour. Agents incrementally learn to depend upon the word of other agents in order to achieve goals. This use of social context in the learning domain of agents is interesting and illustrates one way of how adaptation can be achieved in a multi-agent domain. Carabelea [2001] have modelled traditionally single agent stimulus, i.e events from the environment, as social notions of interaction, i.e requests for actions with the promise of compensation or reward afterwards.

Lin and Debenham [2001] have applied a multi-agent learning approach to the BDI framework in which adaptation occurs as a result of the accumulation of past messages that are received. From this information, a model of other agents is created. This model is then used to help the agent adapt to the foreign agent's strategies. In a sense this is a form of probabilistic adaptation which uses frequency to establish models of other agents, then uses that frequency to exhibit behaviour that reflects the probability of certain actions that other agents may perform. The domain is centered around trust, negotiation and cooperation.

2.3 RoboCupRescue

The problem domain we have chosen to test our BDI learning model is a simplified subset of the RoboCupRescue[3] domain. RoboCupRescue is an international initiative that aims at promoting the development of robotic agents that are capable of responding to natural disasters. The RoboCup Rescue challenge is a branch of the original RoboCup Soccer League

[3] http://www.rescuesystem.org/robocuprescue/

which started in 1993. The aim of RoboCup soccer is to create a team of autonomous humanoid robots that are capable of defeating a real human soccer team by the year 2050. RoboCup Rescue began in 2001 and is modelled after a large earthquake in Kobe, Japan in 1995. Its aim is to create an autonomous team of agents that is capable of entering an area affected by a disaster and to render assistance. Assistance can be in the form of extinguishing fires, un-burying trapped people, clearing road blocks or transporting injured people to hospitals. Our work focuses on extinguishing fires. In achieving this, we have extended the notion of fire extinguishers to represent more than just one retardant.

The way the RoboCupRescue system is organised is via a centralised model with a central 'kernel' that has several sub-systems that connect to it. These sub-systems simulate the various disasters such as road blockages, fires and trapped/injured civilians as well as the agent systems that represent the rescue teams such as the *police* (which clear road blockages), *fire fighters* (which extinguish buildings that are on fire and un-bury trapped civilians) and *ambulance* (which transport injured civilians to nearby hospitals). The kernel acts as the 'hub' where all the information is passed from the sub-systems to be processed by the kernel who's role is to simulate 'time-steps' and to coordinate and simulate the interactions of all the separate sub-systems into a synchronised system as a whole.

Within our domain, we use five different types of retardants, namely: Water (H_2O), Carbon Dioxide (CO_2), Halon, Dry Chemical and Foam. *Water* is the most common fire retardant used on common combustibles such as paper, cardboard and wood. *Carbon Dioxide* is a gaseous retardant that forces oxygen away from a fire and can be used on electrical and oil and solvent fires. *Halon* is also a gaseous extinguisher but is more effective than Carbon Dioxide. *Dry Chemical* retardants are powders that are spread across a fire affected area. Finally, *Foam* retardants work by floating across the surface of a liquid fire and can also be used on common combustibles as well. We have assumed that the weakest retardant is water with the strongest retardant being Halon. The complete order of strength from weakest to strongest retardant is: Water, Carbon Dioxide, Dry Chemical, Foam and Halon.

Chapter 3

Learning in the BDI Framework

At present, the BDI model of agency has very little in the way of learning. Current adaptation in BDI agents is achieved by means of choosing alternative plans according to the current situation. However, there is no generic mechanism by which an agent can store, retrieve and process previously encountered cases. This chapter presents an alternative to the current BDI model in which the agent's history is taken into account. We begin by introducing the general concepts of learning in the BDI execution cycle, then we present a more concrete model and describe the data structures and algorithms we have implemented as part of our executable agent system which we use later as part of our experimental test-bed. This chapter will begin by introducing our learning model followed by how Inductive Learning and Statistical Learning can be integrated into the BDI framework. Finally we will present how we implemented our agent learning system into the JACK Intelligent Agents Toolkit.

3.1 Learning Model

Our work can be seen as an extension of the BDI execution model presented in [Rao and Georgeff, 1995] which can be seen in Figure 3.1. As seen in Figure 3.1 the BDI execution model situates an agent in some world with an initial *state*. The agent is then given a set of *options* upon which to *deliberate*. Once a subset of options is chosen, these options are committed to and placed into the agent's intention stack. Upon execution a new set of events is generated and successful or impossible attitudes are removed from the agent's reasoning cycle.

In modifying the BDI execution model in Figure 3.1, our work aims to achieve historical reasoning through means of belief modification. Hence our contribution to the BDI execution

```
initialise-state();
REPEAT
   options := option-generator(event-queue);
   selected-options := deliberate(options);
   update-intentions(selected-options);
   execute();
   get-new-external-events();
   drop-successful-attitudes();
   drop-impossible-attitudes();
END REPEAT
```

Figure 3.1: BDI Interpreter

```
initialise-state();
REPEAT
   options := option-generator(event-queue);
   [selected-options := deliberate(options, learnt-knowledge)];
   update-intentions(selected-options);
   execute();
   get-new-external-events();
   drop-successful-attitudes();
   drop-impossible-attitudes();
   [history:=update(history, selected-options, environment)];
   [IF ready to learn THEN]
      [learnt-knowledge := learn(history, current-state)];
   [END IF]
END REPEAT
```

Figure 3.2: Learning BDI Interpreter

model is the addition of learning and analogous reasoning into the reasoning process of BDI agents. As seen in Figure 3.1, the algorithm is an extension of the traditional BDI execution cycle with the square brackets indicating the extensions. The learning agent receives an event from the environment and from that event, calculates an applicable plan set which is represented by *options*. From this, the agent *deliberates* on these options but this time it gives consideration to any previously *learnt-knowledge* which acts as an additional layer of information which helps the agent filter the less desirable options from the more desirable ones. The agent then selects a plan from the selected options and *executes* it. Once the agent has completed its task execution, the agent *drops* any successful or impossible goals and updates the *history* with the *selected-options* and *environment*. This creates a record of the state of the world that existed when the agent performed a certain task. This gives the agent a chance to reason about the cause and effect relationships between action and outcome if it wishes to do so. In doing this, the agent's behaviour is reinforced by its past experiences. Before the reasoning cycle ends, the agent must decide whether to exploit its history and *learn*.

The *learnt-knowledge* addition to the BDI execution model in Figure 3.1 can be seen as an optimization of the learning process. This addition arose due to the way in which agent performance degraded over time with respect to execution time and memory resources. Hence, as a way of managing the large increase in past memories, the agent would need to be selective in its choice of what memories to use when reasoning about the past.

When converting the algorithm in Figure 3.1 into an implementation, one may choose to create an agent framework from the ground up or extend an already existing agent toolkit. We have chosen the latter option with JACK as the toolkit. Our framework which is based on Figure 3.1 consists of four major components: the JACK system, the *Learning Formatter*, the *Learning Component* and the *Knowledge Extractor*. Figure 3.3 shows our conceptual model.

The flow of information begins with the *BDI agent*. This agent stores its experiences in the *History* beliefset. When enough history accumulates, *Learning Formatter* converts the History and *Background Knowledge* (provided by the agent designer) into an input suitable for the *Learning Component*. It is important to note that *History* is a parameter to the learning component and by 'enough history' we mean that the number of cases in the history has reached a pre-specified number. This pre-specified number is a parameter itself which stipulates when learning should occur. Learnt data is returned and converted into *Virtual Beliefs* by the *Learning Parser* which translates the learner's output into a form readable by

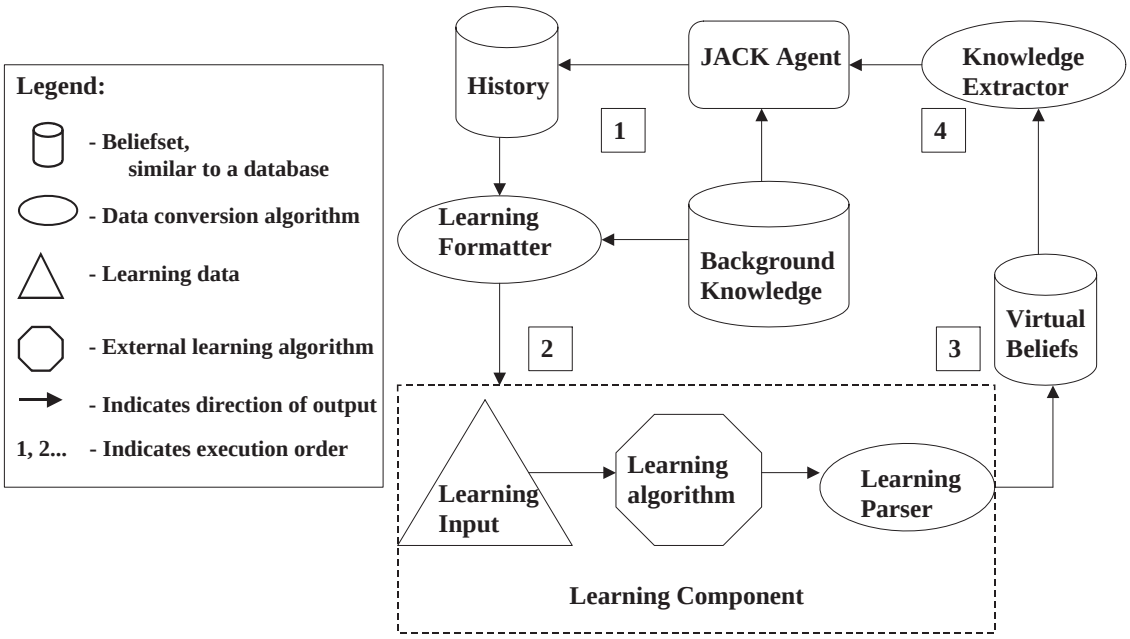

Figure 3.3: BDI Learning Model

the agent. This data is queried by the *Knowledge Extractor*, allowing the agent to reason historically.

The History stored by the agent is a set of tuples containing the *state of the world*, the *outcome* and an *action*. For example, the History tuple ⟨windy, concrete, high, success, water⟩, represents that it was a windy day when the action of applying water was successfully used to extinguish a concrete building burning with a high intensity. In our implementation, to experiment with different search space sizes the *state of the world* varies from 3 to 11 elements.

3.1.1 Learning Formatter

This component represents the first stage of learning. It is here that the historical cases and any background knowledge are combined into a format that the *Learning Component* can understand. The formatted data is then used to produce learning output.

The learning formatter is not tied to any specific format. It is the agent designer who encodes the details of the historical cases, background knowledge and any additional domain specific information. Hence, if another learning mechanism is needed, our model supports

this. The details of how the learning data should be formatted are implemented by the developer. It is this generality that provides a flexible learning model for BDI agents. For the purposes of our implementation we have chosen to use the Alkemy learner and therefore used the input format of Alkemy. A small example of what we used can be seen in Figure 3.1.1. The four main components of the input to Alkemy are the *data declaration*, the *training examples*, the *transformation information* and the *rewrites*. The data declaration defines the variables of the domain with *Weather*, *Buildings* and *Retardants* forming the basis of the tuple *outcome* which represents a state of the environment. The *Class* represents whether the application of a retardant lead to *success* or *failure*. The training examples are basically a mapping of a state and a retardant to an outcome. The transformation information defines the higher order functions such as *negation*, *and*, *tuple projection* and *equality* which are used to represent the hypothesis space. The rewrite system defines the way in which the functions can be combined to help guide the learner through the search space.

3.1.2 Learning Parser

After the learning component has produced a result, the *Learning Parser* converts this output into a format that is understandable by the agent. This is necessary as there exists a wide range of formats that can possibly be returned by any learning algorithm. These formats tend to be specific to the learning algorithm.

The details of how to interpret the output of the learning mechanism need to be specified by the agent designer. For example, if the learner returned functions such as 'Sum(x,y,z)' the agent would have to know what 'Sum()' represented and what parameters it accepts. In general, the Learning Parser is a translation module that sits between the agent and the learning algorithm used by that agent. Figure 3.5 shows some example output of Alkemy which is returned as ASCII text. It states that 'If you use retardant Halon then the fire will be extinguished else the fire will keep burning'. As this is output is just text, the learning agent would have to use the learning parser to convert it into an appropriate data structure for use by the agent. In our implementation we parsed the Alkemy output into an in-memory decision tree which was then traversed by the agent.

A parser had to be created in order for the agent to interpret the learning output. The software used to create such a parser was 'JavaCC'[1]. JavaCC is a Java-based parser creation tool. When given a grammar, JavaCC converts this grammar into a Java program which

[1] https://javacc.dev.java.net/

```
%% -- Data declaration
Steel, Wooden : Buildings ;
DryChemical, Halon : Retardants ;
success, failure : Class ;
Outcome = Buildings * Retardants ;
Individual = Outcome ;
LEARN Recommendation : Individual -> Class ;
%% -- Training examples
Recommendation (Steel, Halon) = success ;
Recommendation (Wooden, Halon) = success ;
Recommendation (Steel, DryChemical) = failure ;
Recommendation (Steel, Water) = failure ;
%% -- Trans information
not : Bool -> Bool ;
not = negation() ;
and2 : (Outcome -> Bool) -> (Outcome -> Bool) -> Outcome -> Bool ;
and2 = conjunction(2) ;
projBuildings : Outcome -> Buildings ;
projBuildings = project(0) ;
projRetardants : Outcome -> Retardants ;
projRetardants = project(1) ;
eqDryChemical : Retardants -> Bool ;
eqDryChemical = equalConst("DryChemical") ;
eqHalon : Retardants -> Bool ;
eqHalon = equalConst("Halon") ;
eqSteel : Buildings -> Bool ;
eqSteel = equalConst("Steel") ;
eqConcrete : Buildings -> Bool ;
eqConcrete = equalConst("Concrete") ;
top : alpha -> Bool ;
top = top() ;
%% -- Rewrites
top >-> and2 (top) (top) ;
top >-> projBuildings . top ;
top >-> projRetardants . top ;
top >-> eqDryChemical ;
top >-> eqHalon ;
top >-> eqSteel ;
top >-> eqConcrete ;
```

Figure 3.4: Example of Alkemy Input

IF and2 (projRetardants.eqHalon) (top) x THEN success ELSE failure

Figure 3.5: Alkemy Output

parses input. Depending on whether the input given matches the original grammar, the program generated by JavaCC will determine whether it is valid or not. The grammar given to JavaCC to construct the parser program can be seen in Figure 3.1.2.

```
ID = [a-zA-Z0-9_]+
IDENTIFIER = {ID}
composite : atom | composite '.' atom | composite '.' '(' atom ')'
                 | '(' composite ')' ;
atom : IDENTIFIER | IDENTIFER arguments ;
arguments : '(' composite ')' | '(' composite ')' arguments ;
```

Figure 3.6

The grammar in Figure 3.1.2 specifies how an Alkemy expression such as:
and2 (projIntensity.eqHigh) (projWeather.eqHot) should be parsed. According to the grammar, *and2* is considered an atom with *(projIntensity.eqHigh)* and *(projWeather.eqHot)* being its arguments. In turn, these two arguments are also treated as atoms, which are separated by the '.' operator.

In using the above grammar, JavaCC allowed the parsing of the decision tree produced by Alkemy into a Java object which was then used by the learning agent. Effectively, JavaCC acted as the bridge between JACK and Alkemy. We would like to note that only the attribute-value part of Alkemy will be used in our work (given this, C4.5 could also have been used).

3.1.3 Knowledge Extractor

The purpose of the *Knowledge Extractor* is to allow the agent to 'make sense' of its learnt knowledge. Depending on what format the learnt knowledge/virtual beliefs are in, the knowledge extractor produces a recommendation by referring to its learnt knowledge to try to find the closest match possible to its current environmental state for a solution and then applying validity checks before using that solution. Given a specific set of data and in some cases a query, the knowledge extractor will return a value that will result in a predicted outcome.

The operation of the *Knowledge Extractor* involves the following steps:

If using Alkemy:

1. Estimate the accuracy of the tree produced by Alkemy
2. If the accuracy is "good enough", that is if the accuracy is above a certain threshold then use the recommendation produced by the tree, else explore

If using Statistical Learning:

1. Using Statistical reasoning from the agent's historical case set to select an appropriate retardant, else explore

In the case where the learning agent is using Alkemy, the agent must be confident that the tree's accuracy is good enough. The way in which the agent determines whether the decision tree is accurate enough is by comparing its perceived accuracy against one of three values: (1) A static threshold (2) A dynamic threshold without analogous reasoning or (3) A dynamic threshold with analogous reasoning. With static thresholding, it is simply the case of comparing the accuracy against a constant value, e.g '0.5'. With the use of dynamic thresholding without analogous reasoning, the threshold is adjusted based on how many times the recommended retardant was able to extinguish previous fires that *exactly* match the current fire. When using dynamic thresholding with analogous reasoning the process becomes slightly more complicated.

The basis behind dynamic thresholding is to calculate the effectiveness of a retardant based on how useful it was in the past. It also ensures that only relevant cases are selected and considered rather than looking at every case in the history set which would include some irrelevant cases.

The formula used to calculate dynamic thresholding is:

$$\text{threshold} = \text{static threshold} - \frac{\text{successful cases} - \text{failed cases}}{2 \times \text{total cases}}$$

As seen above, the successful cases and the failed cases are taken into account with failed cases providing a negative effect to the threshold value. All historical cases are considered to provide a global scaling of the usefulness of retardants. The purpose of the double multiplier

and the 'static threshold' is to maintain the condition that the overall dynamic threshold returned is kept within the range [0.0-1.0].

When searching for past fires that match, the larger the search space, the less likely the agent is to find an **exact** match. Hence dynamic thresholding may not be too effective given a significantly large search space. One approach to solve this issue is to introduce the notion of 'similarity', effectively relaxing the constraint that is imposed by exact matching. The use of the Simile algorithm in conjunction with dynamic thresholding allows the agent to do this.

Analogous reasoning (*Simile*) can be applied to include an additional subset of the total history into the learning process. This analogous filtering of past episodes expands on information which may be related to any search through the agent's virtual beliefs. The benefit of this is that past cases which may be similar but not exactly match may yield relational knowledge which may prove useful. Hence, instead of discarding all fires that are not exactly matching, the Simile algorithm may classify some cases as relevant and hence make some use of fires that would be otherwise thrown away. As an example of the *Simile algorithm*, imagine the agent receiving information about a fire on a mild day. If the agent has *not* encountered this fire before, it would have to explore in order to choose a retardant. But if the agent had encountered a fire with *similar* characteristics with there only being a difference in the weather type, i.e instead of 'mild' as in the current fire the past case had 'hot' weather, then the outcome of the past case can be used to decide whether the same retardant should be used again in the current case. This is assuming that fires on hot days are more difficult to put out than on mild days, knowing what works on a hot day could possibly work on a mild day. We will be discussing the Simile algorithm in Section 3.5.

In the following sections we will be discussing these various learning algorithms in more detail. We will discuss Inductive and Statistical learning within the BDI agent framework, the Simile algorithm, the Sliding Window algorithm and finally the implementation details of our agent learning system.

For the purposes of constructing our fire fighting system, only a subset of Alkemy's functionality was used. Specifically, this involved the use of the *And*, *Not*, *Equality*, *Dot*, *Projection* and *Top* operators.

3.2 Inductive Learning Within the Agent Framework

To predict an outcome from learnt knowledge derived from an induced decision tree, the current state of the world must be given to the *Knowledge Extractor* which then gives a

recommendation. Producing a recommendation from the Alkemy decision tree is done as follows: (1) The Knowledge Extractor scans the higher order function tree to see what values exist for the *retardant* variable. If none are found, then the agent has had no relevant prior experience and will return *unknown* or a default value. In the case of 'unknown' being returned, the agent would then perform exploratory action in order to select a retardant to use. If however, values are found then (2) the Knowledge Extractor will record every unique value into a set along with the value "none-of-the-above" at the end of the list. So for example, if the decision tree had knowledge about the retardants 'water', 'carbon dioxide' and 'halon' then the set would look like: water, carbon dioxide, halon, none-of-the-above. The reason why "none-of-the-above" is added is to compensate for other retardants that do not exist in the tree. This forms a set of *potential* retardants that may prove useful. The next step is (3) where every unique value in the set *potential* is combined with the current state to create a complete query to the decision tree. For each of these complete queries, the Knowledge Extractor uses the decision tree to predict the outcome of using that particular retardant on the current fire. Step (4) is then executed where those retardants for which the decision tree predicts a successful outcome are retained as the tree's recommendation. However, the accuracy of the tree itself is still not known, hence it must be tested. A given threshold of tolerance in the tree's accuracy must be met before any recommendation can be considered reliable.

The *Knowledge Extractor* algorithm can be seen below:

Algorithm Knowledge Extractor (Alkemy)
Input: Current state, S
Alkemy Decision Tree, *tree*
FocusAttribute, *attribute*
Output: A set of recommendations for *attribute* derived from *tree*

Let *potential* be an empty set
Let *results* be an empty set
If agent has seen enough cases **then**
  Traverse *tree* and add every unique value associated with *attribute* into *potential*
  Insert "None Of The Above" as a value for *attribute* into *potential*
  Calculate accuracy of *tree* (Refer to 3.3 below for details)
  If Size of *potential* > 1 **then**

For every element e in *potential* **loop**
Create complete query with e and traverse *tree* using query
Calculate Dynamic Threshold of e
If Accuracy of tree is greater than threshold **then**
Add e to *results*
endfor
Return *results*
Else Explore

As an example, if the agent's decision tree was the same as seen in Figure 2.4 then after the algorithm returns all unique retardant types, the set *potential* would then be Water, Foam. At this point, the set *potential* would have the *None-Of-The-Above* value inserted into it to produce *Water, Foam, None-Of-The-Above*. Step (3) is then executed which systematically combines the current fire state and each retardant in *potential* into a 'state-action' pair that is used as a query to the decision tree. If the state of the current fire is:
$\langle Intensity.Low, Weather.Hot, Building.Steel, Pressure.Average\rangle$
then the first complete query would be
$\langle Intensity.Low, Weather.Hot, Building.Steel, Pressure.Average, Retardant.Water\rangle$.
Querying the decision tree with this query would yield the result 'Fire Extinguished'. Next, 'Foam' would replace 'water' from the previous query, resulting in 'Fire Not Extinguished'. Finally, 'none-of-the-above' would be the retardant added to the current fire state and parsed as a query to the decision tree. This would result in 'Fire Not Extinguished'. Overall, only 'Water' is a useful retardant, provided the accuracy of the decision tree is above the threshold of reliability.

3.3 Calculating The Accuracy Of Decision Trees

Estimating the accuracy of the decision tree is done by checking the tree's predictions against the outcomes of all recent fires that the agent has fought which have **not** yet been used for learning. This collection of *unseen* fires is accumulated over time and whenever the tree is used, its accuracy needs to be tested to ensure that the answer it provides can be considered reliable. In machine learning terms, the decision tree has been given *training* data and therefore must then be given *test* data to verify its accuracy. The outcome of this testing results in a number between 0 and 1. For example, if there are 37 recent fires that have not

yet been learned from and for 32 of them the decision tree correctly predicted the outcome, then the estimated accuracy of the Alkemy tree is $32 \div 37 \approx 0.865$. By 'correctly predicted' it is meant that the decision tree is able to predict the same result given by the environment.

The algorithm for determining the accuracy of the decision tree is:

Algorithm Determines accuracy of the decision tree
Input: Alkemy Decision Tree, *tree*
Unseen History, *unseen*
Output: Accuracy value between 0.0 and 1.0

Let *accuracyCount* be an integer initialised to 0
Let *result* be an empty string
For every tuple t in *unseen* **loop**
Traverse *tree* with t as query and store outcome in *result*
If *result* is equal to outcome of t **then**
accuracyCount := *accuracyCount* + 1
endfor
return accuracyCount / number of tuples in *unseen*

3.4 Statistical Learning Within the Agent Framework

As an alternative to using an inductive learning approach, the Knowledge Extractor can use Statistical learning [Hastie et al., 2001] in the process of making a choice of retardant. The Statistical learning we implemented was a simple tally of how many times a particular retardant succeeded and failed in extinguishing particular fires that exactly match past fires. This provides a numeric measure of how 'useful' a retardant is. When combined with *Simile* (Section 3.5), a greater subset of the historical case set is included into the reasoning process. This equates to an analogous statistical reasoner that is not only concerned with relevant cases for tallying but also takes into account cases which show characteristics that are similar to the current fire.

The way in which we calculate the effectiveness of each retardant is via the formula:

$$\text{effectiveness} = \frac{\text{successes} - \text{failures}}{\text{total}}$$

The retardant with the highest effectiveness is then selected. There are several variants of this depending on whether one considers all past fires, or only past fires similar to the current fire. Note that this simpler mechanism bypasses the learning component depicted in Figure 3.3, since it only requires the agent's history.

The algorithm for Statistical learning can be seen below:

Algorithm Knowledge Extractor (Statistical)
Input: History case set *history*
Current fire state, *state*
FocusAttribute, *attribute*
Output: A recommendation for *attribute* statistically derived from *history*

Let RetardantSuccess, RetardantFailure and RetardantOther be arrays of integers
Set size of RetardantSuccess to number of values *attribute* has
Set size of RetardantFailure to number of values *attribute* has
Set size of RetardantOther to number of values *attribute* has
Initialise all elements of RetardantSuccess, RetardantFailure and RetardantOther to 0
For every tuple t in *history* **loop**
  If t exactly matches *state* and t.outcome == failure **then**
    Increment RetardantFailure indexed relative to the *attribute* value
  Else if t exactly matches *state* and t.outcome == success **then**
    Increment RetardantSuccess indexed relative to the *attribute* value
  Else
    Increment RetardantOther indexed relative to the *attribute* value
endfor
Calculate statistical relevance score for each value using the formula:
(RetardantSuccess - RetardantFailure) / (RetardantOther + RetardantSuccess + RetardantFailure)
If any statistical relevance score is greater than all others **then**
  Return value with highest statistical relevance score
Else
  Explore

The above algorithm requires as input (1) an historical case set, e.g a beliefset (2) the current fire state and (3) an attribute to focus upon, i.e a retardant. From this input, the

algorithm returns a value of *attribute* that is most likely to result in a positive outcome. The algorithm works by recording the number of times 'success' and 'failure' occur for each unique instance of *attribute*. The cases which are considered must be exact matches to the current state, otherwise they are classified as 'other'. All these values are then used to calculate weights for each unique value and the value with the highest weight/success rate is chosen. In the event that all retardants have the same weight, an *explore* command is invoked which then gets the agent to invoke its *Explore* plan.

During our experiments, we noticed that the learning agent would sometimes encounter cases where two or more retardants had the same probability of success. A logical solution would be to select any of those retardants since their probabilities are equal. The algorithm in Figure 3.4 expected that retardants with the highest probabilities be chosen and when all retardants had an equal probability of success, a random selection be made. However, this raises the issue of omitting retardants whose probabilities of success are very close but lower to those retardants most likely to succeed. Hence we decided to develop an algorithm that took advantage of this subtle difference in the success rates of the various retardants. This new algorithm involves modifying the existing statistical algorithm seen above to handle the conditions when there are two or more retardants that have a similar probability of success. In essence, we are returning a *set* of retardants instead of just one retardant.

As an example, consider the situation where four retardants each have their own probabilities of success, say 90%, 89%, 30% and 20%. The modified statistical algorithm could potentially return all or none of the retardants depending upon whether each retardant is equal to or greater than a given *threshold*. It is this threshold which allows multiple retardants to be returned. If in this case the threshold was 50% or higher, the two retardants with probabilities 90% and 89% would be returned. In the cases where all retardants have an equal chance of extinguishing the current fire, a random choice is made.

Algorithm Knowledge Extractor (Statistical with Clustering)
Input: History case set *history*
Current fire state, *state*
Focus Attribute, *attribute*
Cluster difference threshold, $ClusterThreshold$
Output: A recommendation for *attribute* statistically derived from *history* with clustering

Let RetardantSuccess, RetardantFailure and RetardantOther be arrays of integers

Set size of RetardantSuccess to number of values *attribute* has
Set size of RetardantFailure to number of values *attribute* has
Set size of RetardantOther to number of values *attribute* has
Initialise all elements of RetardantSuccess, RetardantFailure and RetardantOther to 0
For every tuple t in *history* **loop**
    If t exactly matches *state* and t.outcome == failure **then**
        Increment RetardantFailure indexed relative to the *attribute* value
    Else if t exactly matches *state* and t.outcome == success **then**
        Increment RetardantSuccess indexed relative to the *attribute* value
    Else
        Increment RetardantOther indexed relative to the *attribute* value
endfor
Calculate statistical relevance score for each value using the formula:
(RetardantSuccess - RetardantFailure) / (RetardantOther + RetardantSuccess + RetardantFailure)
Round off statistical relevance scores to 4 decimal places
If any 2 or more statistical relevance scores are equal **then**
    Sort all statistical relevance scores from highest to lowest
    For every statistical relevance score r **loop**
        If highest statistical relevance score - $r \leq ClusterThreshold$ **then**
            Add r into result cluster *cluster*
    endfor
    return Random retardant from *cluster*
Else
    Explore

As seen above, the Statistical algorithm has been modified to include clustering functionality that is focussed upon the statistical relevance scores. This new algorithm is identical to the original Statistical algorithm up until the point where the statistical relevance scores are compared against each other. As with the original Statistical algorithm, if any retardant has a clearly higher statistical relevance score then that retardant is returned. Also if all retardants have the same score then exploration is conducted. The new functionality entails the clustering of retardants that are within a certain *cluster threshold*. This value is given as a parameter and represents the maximum threshold of inclusion into the cluster. Each

retardant's score is subtracted from the highest scoring retardant with the difference being compared to the threshold. If the difference is less than or equal to the threshold then the retardant is included into the cluster, otherwise it is ignored. The final recommendation given by the algorithm is randomly chosen from the cluster. The value of the cluster threshold itself was determined during experimentation and is explained in more detail in Chapter 4.

3.5 The Simile Algorithm

The *Simile algorithm* that we developed is an analogous reasoner that allows the agent to relax its 'exact matching' constraints thereby allowing more cases to be considered. The hypothesis is that by including more cases, the predications can become more accurate. Not all cases can be added due to the fact that some states are incomparable.

When analogous reasoning is used, the following scenarios are used:

1. Past fires that were harder and succeeded.
2. Past fires that were easier and failed.
3. Past fires that were easier and succeeded.
4. Past fires that were harder and failed.
5. Past fires are incomparable

Scenarios (1) and (2) are considered by the learning agent while scenarios (3) and (4) are discarded. The reasoning behind using (1) is because if a past fire was harder than the current fire and it was successfully extinguished, then it can be assumed that the current fire, which is easier by comparison, will also be extinguished by the same retardant. With (2), if a past fire was easier and was not extinguished with a particular retardant, then that retardant is definitely not going to work on the current fire which is harder therefore the agent can eliminate that retardant from being recommended. The reason why scenarios matching (3) are not considered when searching for retardants to recommend is because it only covers cases that were easier, not equal to or harder than the current fire. For example, if water is known to extinguish fires of lower difficulty than the fire currently being fought, then the agent can not be certain that the same success will happen with a more difficult fire. Hence, these kinds of cases are not added to the set that is used to calculate the usefulness of a retardant. With (4), a similar principle as in (3) applies where the past case

was harder and failed. This case only describes failure at harder cases, yet says nothing about the outcome with easier cases, i.e the current fire being fought. Scenario (5) describes cases where within the same comparison, some attributes are easier while others are harder. This renders cases 'incomparable' as there is no fire that is clearly easier than the other. As an example, consider the case where one fire has an easier 'Weather' attribute but at the same time has a harder 'Building Type' attribute. In relation to these rules we wish to acknowledge that defining what is 'easier' or 'harder' may not always be an easy task and as a result, in our implementation the agent had prior knowledge of what fire attributes were 'easier' and 'harder' etc. Therefore this can be considered as a form of supervised leaning in which an expert has already labelled what fire attributes are easier and harder.

To demonstrate, suppose the agent is fighting a fire in *Hot* weather where the building is made of *Wood* and the fire is burning with a *High* intensity. A previously fought fire that was on a *Mild* day, involved a *Steel* building and was of *Medium* intensity would be considered **easier** then the current fire. If a particular retardant was unsuccessfully used on the previous, easier, fire then the simile algorithm will reason that the retardant in question is probably a bad choice for the current, harder, fire as well. We will discuss in more detail the notion of difficulty in Section 3.8.

The *Simile* algorithm can be seen below:

Algorithm Simile
Input: Current state, S
A historical case, h
Output: Indication of whether current fire state S is: 'harder', 'easier', 'incomparable' or 'same'
Initialise counters *easy* and *hard* to 0
For every attribute a in h **loop**
  If attribute a is easier than corresponding attribute in S **then**
    easy := easy + 1
  Else if attribute a is harder than corresponding attribute in S **then**
    hard := hard + 1
endfor
If $easy > 0$ and $hard = 0$ **then Return** 'easier'
Else if $hard > 0$ and $easy = 0$ **then Return** 'harder'
Else if h is equal to S **then Return** 'same'
Else Return 'incomparable'

As seen above, the Simile algorithm returns one of four possible values which indicate that a given past case is either 'easier', 'harder', 'same' or 'incomparable'. For an historical case to be considered 'easier' there must be at least one fire attribute that is 'easy' with no attributes classified as 'hard'. The reasoning behind this is to ensure that the fire states are comparable. If one or more variables have an inverse difficulty relationship with the other variables in the same fire, then this violates the rule that all variables must be either all harder or all easier and hence are considered 'incomparable'.The same rule applies to the 'harder' case where every attribute in the past fire must have a higher difficulty than the current fire. This is because of any unknown relationships that may exist between any of the variables in a given fire state. Only by knowing that *all* variables are *greater than or equal to* or *lower than and equal to* the current fire can any conclusive judgement be made. The way in which the Simile algorithm is used with the Alkemy and Statistical learners is during the phase where past cases are gathered for creating learning examples, Simile is called to add any extra cases that it may see as beneficial to the learner. Given this new, potentially larger case set the learner proceeds to produce a recommendation.

3.6 The Sliding Window Algorithm

Our investigation into the issue of pruning past cases to increase the efficiency of the agent learning system resulted in what we call the *Sliding Window* algorithm. This algorithm is adapted from the Sliding Window algorithm proposed by Comer [1995] however instead of being used in a network packet transmission context we use a similar concept to restrict the amount of past cases that are passed in as training examples to the learner. When using the Sliding Window algorithm, an integer x representing the 'size' of the window is given to the agent which then proceeds to only give 'x' number of past cases to the learning module. For example, if the agent is told to use a window size of 300 and is told to learn every 1000 fires, once the 1000th fire is encountered the agent would give only the last 300 fires fought to the learning module to use. Without the Sliding Window algorithm, the agent would pass all 1000 fires to the learning module and when the next learning period occurs at 2000 fires, all 2000 fires are given to the learning module. Hence the Sliding Window algorithm prevents the agent from using its entire history for learning.

3.7 Exploration

When the learning agent is presented with a set of recommendations by the learning module, it has the choice of either accepting that this information is of a reasonable accuracy and therefore use it or it may reject the validity of the information and opt for an alternative way of selecting a retardant. In the latter case, the agent may utilise the process of *exploration* or the use of default rules in order to find a suitable retardant. The reason why exploration is needed is because the accuracy of learnt information may not be over the threshold value of trust in the learning module. Essentially, exploration consists of subtracting different sets of retardants and using the resulting set as the basis for selecting a potentially useful retardant. The way in which exploratory retardant selection is implemented is done in three stages:

1. Selecting retardants that have never been used
2. Selecting retardants that have been used but never recommended
3. Randomly selecting retardants from the set of all known retardants

The above list represents the order in which the agent performs exploration. Starting with (1), the agent chooses retardants which it has never used at all. This is achieved by subtracting the set *Seen* from set *Full* where *Seen* represents the list of retardants that the agent has used and *Full* represents the full list of all retardants available to the agent. If the resulting set is empty, stage (2) is then executed which involves the subtraction of set *Potential* from set *Seen* where *Potential* represents the set of retardants that occur in the decision tree. By subtracting *Potential* from *Seen*, we get the set of retardants that have been used but do not appear in the decision tree[2]. If this still results in an empty set, stage (3) is executed where a purely arbitrary retardant is chosen from the 'Full' set.

The algorithm used for exploration can be seen below:

Algorithm This produces a set used for the exploration of values
Input: Full set of values, $full$
Set of used values, *seen*
Set of useful values, *potential*
Output: A value, i.e a Retardant, to be used by the agent

[2]Due to the fact that there are delays in learning, there may be retardants that have been tried from Stage (1) but are not yet in the decision tree.

Let Set1 be an empty set
Let Set2 be an empty set
Set1 := $full$ - $seen$
Set2 := $seen$ - $potential$
If Set1 is not empty **then Return** random element from Set1
Else if Set2 is not empty **then Return** random element from Set2
Else Return random element of $full$

3.8 Implementation of Agent Learning System

Now we will show the details of the implementation of our agent learning system. Overall, our system consists of a JACK agent system which is Java- based combined with additional Java code to form the core functionality of the agent. The learning components are implemented in C++ and Java which are combined with a Java-based parser tool to convert the learning output into a format suitable for the learning agent.

The fire fighting system we developed, *Halon*, is a composite of two separate systems, namely *JACK* and Alkemy. The integration of these two systems required a third component, JavaCC.

System Overview

The experimental system we have developed, *Halon*, was written using the JACK intelligent agents toolkit. It consisted of two agents, an *Oracle* agent and a *Fire Fighter* agent. A diagram of the system can be seen in Figure 3.7.

The order in which the messages are sent is read from left to right. Starting at the left, there is a *Fire!* event which initiates the sequence of messages. Once received, the Fire Fighter agent uses the learning model in Figure 3.3 to select an appropriate retardant. Once a *retardant* is chosen, it is sent back to the *Oracle* agent which then decides whether that choice of retardant was successful or not. This is done via a rule-set as outlined in Figure 3.8. Once an *outcome* has been determined, the *Oracle* agent sends an event back to the *Fire Fighter* agent with an outcome of success or failure. This outcome is recorded by the Fire Fighter agent in its *Historical Knowledge*. The Oracle agent can be seen as a simplified replacement to the RoboCup Rescue simulator.

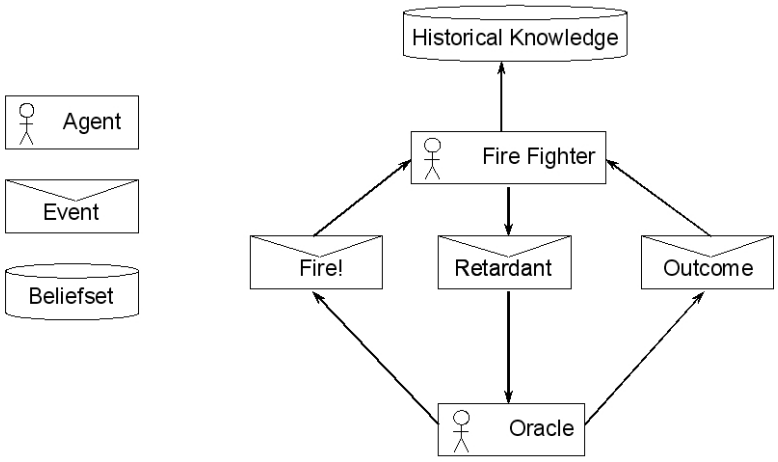

Figure 3.7: System Overview

Fire Fighting Agent

The Fire Fighting agent embodies the data structures and algorithms of particular interest. It is this agent that learns and does historical reasoning. It is given a set of fires to fight, with each fire requiring a single retardant to be suggested that will extinguish the fire.

The fire fighting agent is responsible for the following tasks:

- Receiving a *Fire* event
- Invoking a learning algorithm
- Reasoning about past cases with respect to choosing an appropriate retardant
- Sending a retardant choice to the Oracle agent
- Receiving an outcome from the Oracle agent
- Recording the state, retardant choice and outcome into an Historical beliefset

A single run in the system is started by a *fire* event. By 'run' we mean a sequence of fires, i.e 100, that are fought by the learning agent. The event represents a burning building with various properties. Every fire event is fought by the fire fighter, in which the response is a recommendation of a single fire retardant. As time passes, the fire fighting agent accumulates 'experience' which in turn allows it to recommend more effective retardants.

Once a retardant has been chosen and an outcome has been returned by the Oracle, the fire fighter agent records the environmental state, the retardant and the outcome. These three variables form the history from which is learnt from.

The main data structures that are utilised by the Fire Fighter agent are:

- Background Knowledge
- Historical Case-set
- Learnt Knowledge-set

Historical Case-set

The Historical Case-set is the repository of all past events the fire fighting agent has encountered. It is this data structure that is of primary focus during learning. In general, the format of a historical case is given by: <State, Action, Outcome>. An example of an historical record can be: <[Fierce, Concrete, Rain, Large, Asbestos], Carbon Dioxide, fire extinguished> which can be translated to "*A fierce fire burning in a large concrete building with asbestos insulation on a rainy day was successfully extinguished with carbon dioxide*". This data structure is implemented as a JACK beliefset.

When learning, the agent uses the Historical Case-set as a source of training data. However not all historical cases are necessarily considered if the Simile or Sliding Window algorithms are utilised. When Simile is used, past cases that are normally not considered may be used while with the *Sliding Window* algorithm, only a fixed size of the history is considered, potentially speeding up the process of past case reflection for the learner.

Learnt Knowledge-set

The learnt knowledge set is what stores the outcome of any learning done by the fire fighting agent. This data structure can be considered *domain specific* or abstract as the agent designer must specify the format of the agent's learnt knowledge by specifying how the Learning Parser will convert the 'raw' output of the learner into an agent readable data structure. Depending on what the format of the output is, whether it is a decision tree or a rule-set, the designer must encode the specified format for the agent to manipulate. In the work presented, we implement two separate formats the learning data could take: one for Alkemy and another for Simile. In the case of Alkemy, the learnt knowledge-set was implemented as a binary

decision tree while a JACK beliefset was used to implement the learnt knowledge for the Simile algorithm.

This model has the advantage of being generic whereby new learning mechanisms can be added to an agent without having to alter the agent itself, it is merely an interface by which learning can be easily attached.

Plan-set

The Fire Fighter agent was implemented with a total of four plans, namely the *Explore*, *Extinguish*, *Learn* and *Record-History* plans.

The *Explore* plan was used by the agent to implement explorative behaviour. Specifically, it was used whenever the learning agent had no clear choice of retardant to use either because the learning algorithm could not recommend such a value or because the confidence in the agent's experience was too low.

The *Extinguish* plan is where all the main functionality of the system is encoded. The main algorithms of the system are located in this plan such as when to learn, what learning algorithm to use and when to record historical cases. In a sense, the main execution engine of the agent is embedded within this plan which dictates whether inductive or statistical learning should take place and when it is time to learn, it invokes the *learn* plan.

The *Learn* plan is what implements the parsing from historical cases into learning specific input, namely an Alkemy specification file. Every historical case the agent has stored is converted into training data for Alkemy. Once these cases have been seen by Alkemy, they are moved to another beliefset called *Seen* which separates them from new cases which have not been seen by Alkemy. The reason for this is to be able to use unseen cases to test the accuracy of the decision tree returned by Alkemy.

The *Record-History* plan contains the functionality that stores past fire fighting instances into the *History* beliefset. Its purpose is simple yet is one of the most important as without any historical information, there would be no historical reasoning.

Oracle Agent

The *Oracle* agent represents the environment which randomly generates fires for the fire fighter agent to fight and provides feedback to the fire fighter agent once it has made its choice of retardant. The Oracle agent is responsible for the following tasks:

- Randomly generating a fire state and sending that state to the Fire Fighting agent

- Receiving a retardant recommendation from the Fire Fighter
- Determining whether that retardant recommendation was able to successfully extinguish the current fire
- Sending an outcome result to the Fire Fighter agent

Background Knowledge

The *Background Knowledge* represents prior knowledge. This is given by the agent programmer. Such assumed knowledge used in our learning system includes variables such as *Building Type* and different types of *Weather* conditions. An example of the input file that is read in by the Oracle agent for the construction of its background knowledge can be seen in Figure 3.8.

```
11
Retardant Water CO2 Halon DryChemical Foam
Intensity Fierce 4 IHigh 3 Medium 2 Low 1
BuildingType Wood 5 Steel 3 Concrete 1 Brick 2 Bamboo 4
Weather Hot 5 Mild 3 Windy 5 Rain 0 Overcast 2
Humidity High 1 Medium 2 Low 3
WallType Plaster 4 Wood 3 Steel 1 Plastic 2
FloorType Ceramic 0 Carpet 2 Wood 3
Contents Cardboard 5 Electricals 2 Paper 5 Upholstry 4 None 0
SizeofFire Enormous 4 Large 3 Medium 2 Small 1
SizeofBuilding Enormous 4 Large 3 Medium 2 Small 1
FireSystem Sprinklers 2 SafetyDoors 3 Asbestos 1 None 5
AirDucts Yes 3 No 0
0 12
13 20
21 37
38 50
51
```

Figure 3.8: Format of Oracle's Background Knowledge

The format of the Oracle's background knowledge begins with an integer, in this case '11' which represents the number of variables which collectively represent a fire state. For example, for any given fire that is fought by the learning agent, a fire will consist of a tuple containing all eleven variables. The second line, 'Retardant' specifies the different retardants

that are available to the learning agent for the purposes of extinguishing fires, i.e 'Water', 'Carbon Dioxide' etc. The eleven lines following that represent the various attributes of a fire where each possible value is followed by an integer. These numbers represent the 'difficulty' score of that value where the higher the value, the more difficult it is. For example with the line beginning with 'Weather' we can see 'Hot 5' and 'Windy 5'. This means that both hot and windy weather are the most difficult weather conditions encountered by the learning agent. At the same time 'Rain 0' shows that rainy conditions are the least difficult of weather conditions. The lines starting from '0 12' onwards represent the various tiers of difficulty where some retardants may not be as effective. The way in which these tiers are used is by adding each individual score for every fire attribute, i.e the 'Intensity', 'Building Type', 'Weather' etc, we arrive at a value that represents the overall difficulty of a fire. This value is then matched to see which difficulty tier it belongs to which in turn dictates which retardants are going to extinguish that fire. For example, if the overall difficulty of a fire is 28, then that fire would fit into the third tier of difficulty at which point only certain retardants would be effective at putting out this type of fire.

Oracle Agent's Plan-set

The Oracle agent was implemented with a total of 2 plans, namely the *Start-Fires* and *Determine-Outcome* plans.

The *Start-Fires* plan is responsible for the creation of fires for the fire fighter agent to fight. The process of creating fires is done via two step: 1) Reading in an input file that contains the variables and their associated difficulties and 2) the generation of random numbers, one for each variable to create a fire with various states.

The input file that is read in by the Oracle agent can be seen in the 'Background Knowledge' section. The generation of random numbers is done via the Java *Random* class. This produces a pseudorandom series of numbers that is normally distributed. The random seeds are changed for every experiment and also allows our experimental results to be re-created.

The *Determine-Outcome* plan is used by the Oracle agent to simulate the effects that the Fire Fighter agent has on the environment. Essentially, this plan emulates the part of the environment which responds to the various retardants the Fire Fighter selects. The rules that the Oracle agent uses to determine success and failure can be seen in Figure 3.8.

The numbers in the above rule-set represent the ranges of 'difficulty' of the fires and how

```
<= 12 ALL Retardants work
13-19 EVERYTHING BUT WATER works
20-29 CO2, Foam, Halon work
30-37 Foam OR Halon works
38+ ONLY Halon works
```

Figure 3.9: Oracle Agent's rule-set for determining success

each of the five retardants affects each range. If the difficulty of a fire is less than or equal to 12, then any retardant chosen will result in the fire being successfully extinguished. For example, using the top ranges in Figure 3.8, if the difficulty of the fire is between 13 and 19 inclusive, then any retardant except water will work. As the ranges and therefore difficulty increase, fewer retardants are able to extinguish the fire. At the end of the difficulty spectrum is 38+ where only the most powerful retardant Halon will work.

During initial experimentation, we discovered that the rule-set shown in the top of Figure 3.9 may have not created a difficult enough domain. In other words, we believed that the difficulties of the fires produced could have been more realistic. This conclusion was derived through the 60% success rate of the control experiments in which no learning was used. These control experiments used a random selection of retardants to extinguish fires.

The modification to the rules involved the addition of cases where only one retardant results in success while the other remaining retardants result in failure. These exceptional cases are derived from the highest value of each tier from the Oracle rules. For example, in the second tier of the original Oracle rules in Figure 3.9 we have the range *13-19* where every retardant **except** for *Water* results in success. The addition of exceptions would change this rule to have the range *13-18* where every retardant **except** for *Water* results in success with the highest value of *19* having a restrictive effect whereby only *Dry Chemical* results in success while all others fail. In Chapter 4, we will see that these exceptions did make a difference by increasing the difficulty of the problem domain. Figure 3.10 shows a revised version of the Oracle rules.

```
<= 11 ALL Retardants work
12    Only Water works
13-18 EVERYTHING BUT WATER works
19    Only DryChemical works
20-28 CO2, Foam, Halon work
29    Only CO2 works
30-36 Foam OR Halon works
37    Only Foam works
38+   ONLY Halon works
```

Figure 3.10: Oracle Agent's exception rule-set for determining success

Chapter 4

Experiments

In this chapter we describe and present our experiments and results. We will begin with a discussion of our experimental goals followed by a description of the experimental process. We will then present the findings of each of our research questions and discuss the outcomes and implications of our results.

4.1 Experimental Goals

Experiments were conducted within the fire fighting domain to answer the following research questions:

1. What type of learning should the agent use?
2. How can an agent assess the accuracy/reliability of learning algorithm output?
3. Can the use of analogous reasoning improve Statistical Learning?
4. Does the placing of thresholds on Statistical Learning improve accuracy?
5. What is the effect of pruning the history?
6. When should an agent apply learning algorithms to stored history?
7. What effect do domain characteristics have on learning?

4.2 Experimental Procedure

To explore and validate our experimental goals, we designed and conducted a set of experiments. These experiments were run on a single Pentium 4 2.8Ghz CPU with 1Gb of DDR memory. The operating system used was a distribution of Linux, Fedora Core Version 2.0. The software used consisted of three main components: The JACK Intelligent Agents Toolkit, the learning component (Alkemy or Statistical learning) and JavaCC. The JACK learning agent would periodically call a learning algorithm which returned learnt output. If Alkemy was used, the learnt Alkemy output would then be parsed using a pre-compiled grammar created by JavaCC, a freeware parser generator. This allowed the data to be converted into a more agent-readable form. In general, all experiments were run by passing in certain command line parameters:

- How many fires to fight
- When learning would occur
- What type of learning to use (if learning)
- Dynamic threshold usage
- Search space size
- Whether to use the Simile algorithm
- Sliding Window size (if used)

The output of all our experiments was a series of 'success' or 'failure' results. We measured the overall success of the learning agent in terms of the number of successfully extinguished fires and how long it took the agent to achieve such an accuracy. With the graphs presented in this section, we display the performance of the learning agent in blocks of 100 fires. This means that every marked point on a graph is a calculated average success of the *last* 100 fires across all 50 runs. With regards to the complexity of our domain, this was varied through the use of two space sizes: a domain with 576,000 different possibilities and another with 2,304,000 different possibilities. This was done by representing the environment as a tuple of variables $<v_1, v_2, v_3, ...v_n>$ where: $3 \leq n \leq 11$. Every variable has between two to five possible values each with its own 'difficulty' score. Values are represented as strings while difficulty scores are represented as integers. The difficulty score is used as part of a ranking

scheme that converts symbolic fire states into a numeric representation. This is done to allow us to easily vary the complexity of the domain. These were used to test how a complex search domain affected the agent's learning. To test whether the results are of any statistical significance, we used a standard **two-tail T-test**. As usual, we consider a p-value of 0.05 or less to indicate statistical significance.

Each experiment was run 50 times, where a single 'run' consisted of 3000 fires. Each run was initialised with a different random number generator seed so as to vary the domain in a controlled manner. The results seen in our graphs are the average taken over those 50 runs unless indicated otherwise.

4.2.1 Measuring Performance

The performance of the agent learning system is measured by the percentage of fires extinguished over a given set of fires. The fire fighting agent has no past experience when it begins fighting fires. The performance of the various learning algorithms is measured in terms of both the accuracy and the time taken to return a recommendation.

The tool we used to record our timings was the *time* tool in Linux. It produces 3 separate values per run of the tool (1) the 'real' time (2) the 'user' time and (3) the 'sys' time. The *real* time represents the total time elapsed for the system from the moment the learning system is started to when it is terminated. The *user* value represents how much actual CPU time was taken up by user instructions. The *sys* value represents the system time, the CPU time taken up by the kernel. As we are only concerned with CPU and system cost, our comparisons will consist of comparing the sums of the *user* and *sys* times.

4.3 Types of Learning Used

In relation to the research question 'What type of learning should the agent use?', the two learning algorithms we wish to test in our experiments are Inductive learning, through the use of Alkemy, and Statistical learning. As this is a broad question in which other additional algorithms (such as Simile or Sliding Window) may affect the outcome of our results, we will present these various algorithms in their own subsections as part of this section. We will begin by discussing the general differences between Alkemy and Statistical learning followed by a subsection on the effects of dynamic thresholding on Alkemy, this will then be followed by a subsection on the Simile algorithm and finally by a subsection on Statistical clustering.

The general procedure used to produce our results was to set the domain complexity

(either 576,000 or 2,304,000), select the learning algorithm (either Alkemy, Statistical or no learning), any additional set of reasoning algorithms (Dynamic Thresholding, Simile, Clustering, Sliding Window), the frequency of learning and finally how many fires to fight. As part of our initial experimentation we used both domain sizes with Alkemy, Statistical learning and no learning. Dynamic thresholding was used with Alkemy while Clustering and Simile were used with Statistical learning. The frequency of learning was set to learn every 100 fires and the Sliding Window algorithm was deactivated. These results can be seen in Figures 4.1 and 4.2. The control in our experiment will be to have no learning at all, that is, the agent would fight fires without the aid of any learning ability. In effect, this made the agent select arbitrary choices in retardant instead of trying to learn an appropriate retardant. This allowed us to gauge the effectiveness of different types of learning within the fire fighting domain. As seen in Figures 4.1 and 4.2, this resulted in the lowest accuracies of about 56% and 58% respectively.

If we compare the results from Figure 4.1 and Figure 4.2 we can see that there are distinct differences in the accuracies of all our learning algorithms. The most dramatic difference between the two figures is that the simpler domain in Figure 4.1 doesn't appear to be difficult enough for any differences in predictive accuracy to exist. The main reason for these differences is because of the **exceptions in the rule-set** used to determine success which created a non-linear effect on the success rate of the more powerful retardants. In particular, by removing the case where Halon was effective on all types of fires we can see that more complicated rule-based learners perform better than statistically based learners. In addition to this, in very simple domains where no exceptions exist it is clear that many learners may reach high levels of accuracy with little difference occurring between those learning algorithms. As a result of the simpler domain being too simple, for the remainder of this chapter we will only be dealing with the more complicated domain with exceptions.

The behaviour of the learners is interesting in which Alkemy appears to slowly climb in predictive accuracy throughout the experimental runs. In contrast, the Statistical learner rises to a peak of 68.75% accuracy after fighting 700 fires and roughly remains at that accuracy for the entirety of the experiments. The rate of increase in accuracy of Alkemy is approximately 1% for every 100 fires fought.

Although Alkemy extinguishes more fires than Statistical learning by an average of 10% in the more complex domain, this comes at a time cost four times greater than that of Statistical learning. The Statistical method out-performs Alkemy in the simpler domain, highlighting the fact that complex and powerful learners such as Alkemy are not always the best choice.

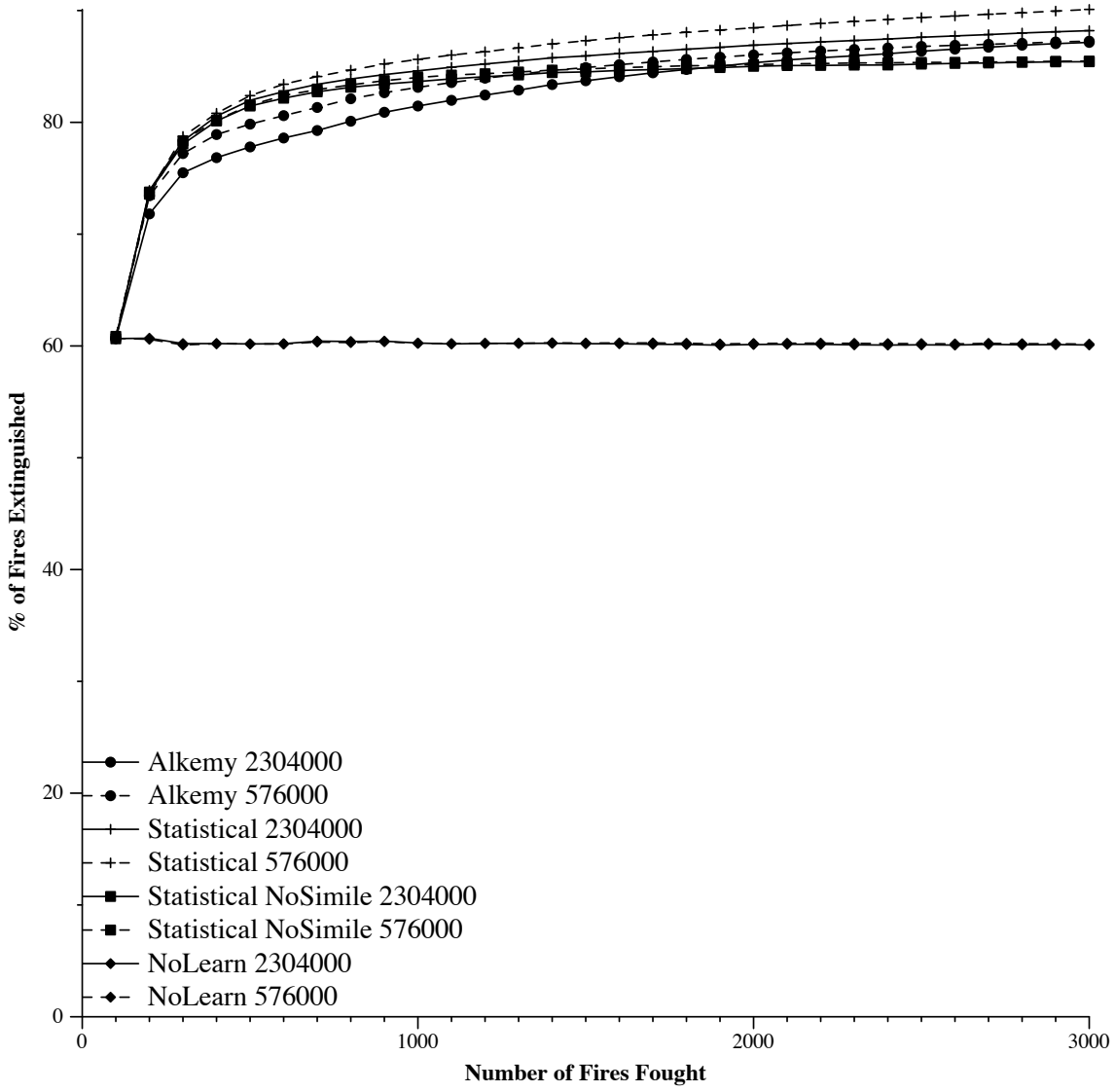

Figure 4.1: Experimental Results for Simple Domain Without Exceptions

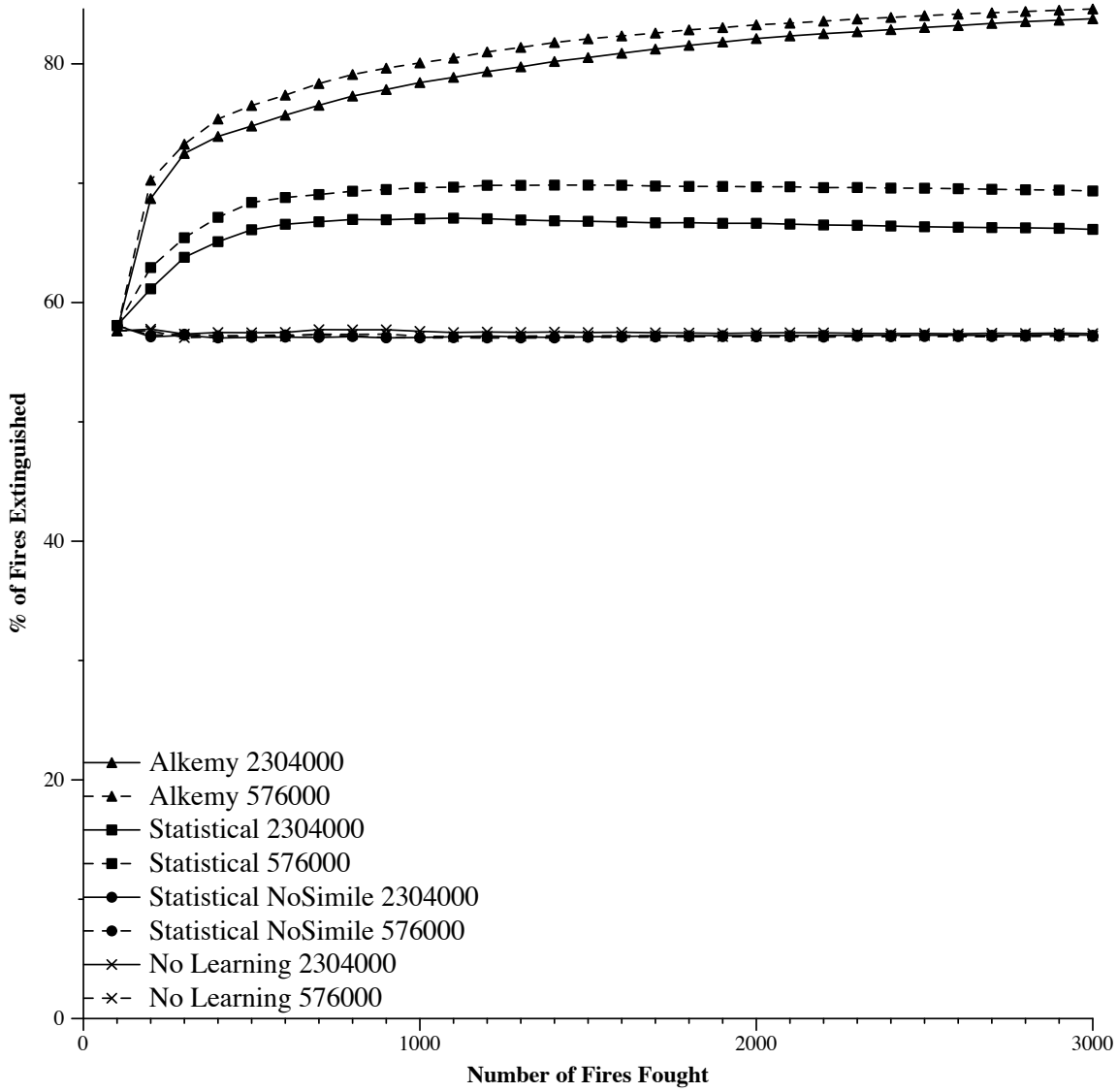

Figure 4.2: Experimental Results for Complex Domain With Exceptions

All learning algorithms except for Statistical without Simile in both search space sizes in Figure 4.2 had a p-value less than 0.0005 when compared against no learning. This implies that learning does make a difference to the agent's performance except in the smaller domain size with Statistical learning without Simile.

The T-test analysis of Alkemy against Statistical learning revealed that none of the results within the simple domain were statistically significant[1]. However, the T-test analysis of the more complex domain with exceptions revealed that all experimental results were statistically significant. We compared all learning algorithms against each other in both search space sizes and all p-values were below the threshold of 0.05[2].

The timing results for Figure 4.2 showed a significantly large difference between the Alkemy and Statistical learning algorithms. As seen in Table 4.1 an Alkemy run takes approximately 45 times longer than a Statistical learner run. As a bench mark, we deactivated all learning algorithms and left the agent to arbitrarily choose a retardant. This took only 1.5 minutes to randomly fight 3000 fires. These timing results clearly show that inducing decision trees using Alkemy takes far longer than Statistical learning.

Algorithm	Time taken in large domain	Time taken in small domain
No Learning	1m 34.58s	1m 35.8s
Alkemy	44h 55m 52.77s	53h 58m 12s
Statistical with Clustering	60m 23.12s	60m 55.18s
Statistical without Simile	60m 55.35s	61m 7.46s

Table 4.1: Times taken for Exception domain

The most interesting point is that Alkemy in the smaller search space takes more time than in the larger search space. This counter-intuitive outcome, may be caused by the fact that adding exceptions affects smaller domains more than it does larger domains. Hence, the fewer combinations there are, the more influence exceptional circumstances have on the time

[1] Alkemy 2304000 vs Statistical 2304000 = 0.132; Alkemy 2304000 vs Statistical w/o Simile 2304000 = 0.596; Alkemy 576000 vs Statistical 576000 = 0.117; Alkemy 576000 vs Statistical w/o Simile 576000 = 0.901; Statistical 2304000 vs Statistical w/o Simile 2304000 = 0.279; Statistical 576000 vs Statistical w/o Simile 576000 = 0.082

[2] Alkemy 2304000 vs Statistical 2304000 < 0.0005; Alkemy 2304000 vs Statistical w/o Simile 2304000 < 0.0005; Alkemy 576000 vs Statistical 576000 < 0.0005; Alkemy 576000 vs Statistical w/o Simile 576000 < 0.0005; Statistical 2304000 vs Statistical w/o Simile 2304000 < 0.0005; Statistical 576000 vs Statistical w/o Simile 576000 < 0.0005

taken to induce a decision tree. Having more exceptions than those outlined in Chapter 3 leads to a degradation in time performance for the larger search space.

Overall, we can conclude that Alkemy is more accurate than Statistical learning however this comes at a cost in time. If accuracy is needed with real-time constraints then either Statistical learning should be used or Alkemy should be run less frequently. It appears that in simpler domains any learning algorithm can be used to great effect when compared to no learning however in more complicated domains other, more sophisticated approaches must be used. For a more detailed discussion of how domain features affect the performance of the learners tested, refer to Section 4.6.

4.3.1 Assessing The Accuracy Of Learnt Data

Our goal for this research question is to assess the effectiveness of the application of thresholds for the purposes of ensuring the reliability of learnt data. The way in which thresholds work is to place a boundary by which agent can say "If accuracy of the output given by the learner is greater than or equal to the threshold, then I believe that the learner is reliable, therefore I will use it". By setting certain thresholds the agent's effectiveness can be altered to improve its performance.

Dynamic thresholding is used **only with the Alkemy learner** as it is a technique for adjusting the acceptance threshold for a decision tree. Statistical learning has a similar technique which will be discussed in Section 4.3.3. The experiments conducted for Alkemy involved setting static and dynamic thresholds with the static threshold being set at 0.5 or 50%. This means that after Alkemy returns a decision tree, it is tested with a subset of the agent's history and if the decision tree predicts 50% or more of the outcomes correctly then the tree's prediction is used, otherwise a random choice is made. With dynamic thresholding on, a previously static 50% threshold may vary from anywhere between 0% to 100% depending upon the specific retardant and how successful it was. For the purposes of analysis, we will be comparing Alkemy with a static threshold to Alkemy with dynamic thresholding so as to gauge how dynamic thresholding allows an agent to assess the accuracy/reliability of learning algorithm output.

As seen in Figure 4.3, dynamic thresholding makes little to no difference to the predicative accuracy of Alkemy. A T-test analysis comparing Dynamic Thresholding against no Dynamic Thresholding shows that the results are not statistically significant. The high p-values of 0.988 for the larger search space and 0.984 for the smaller search space show that dynamic

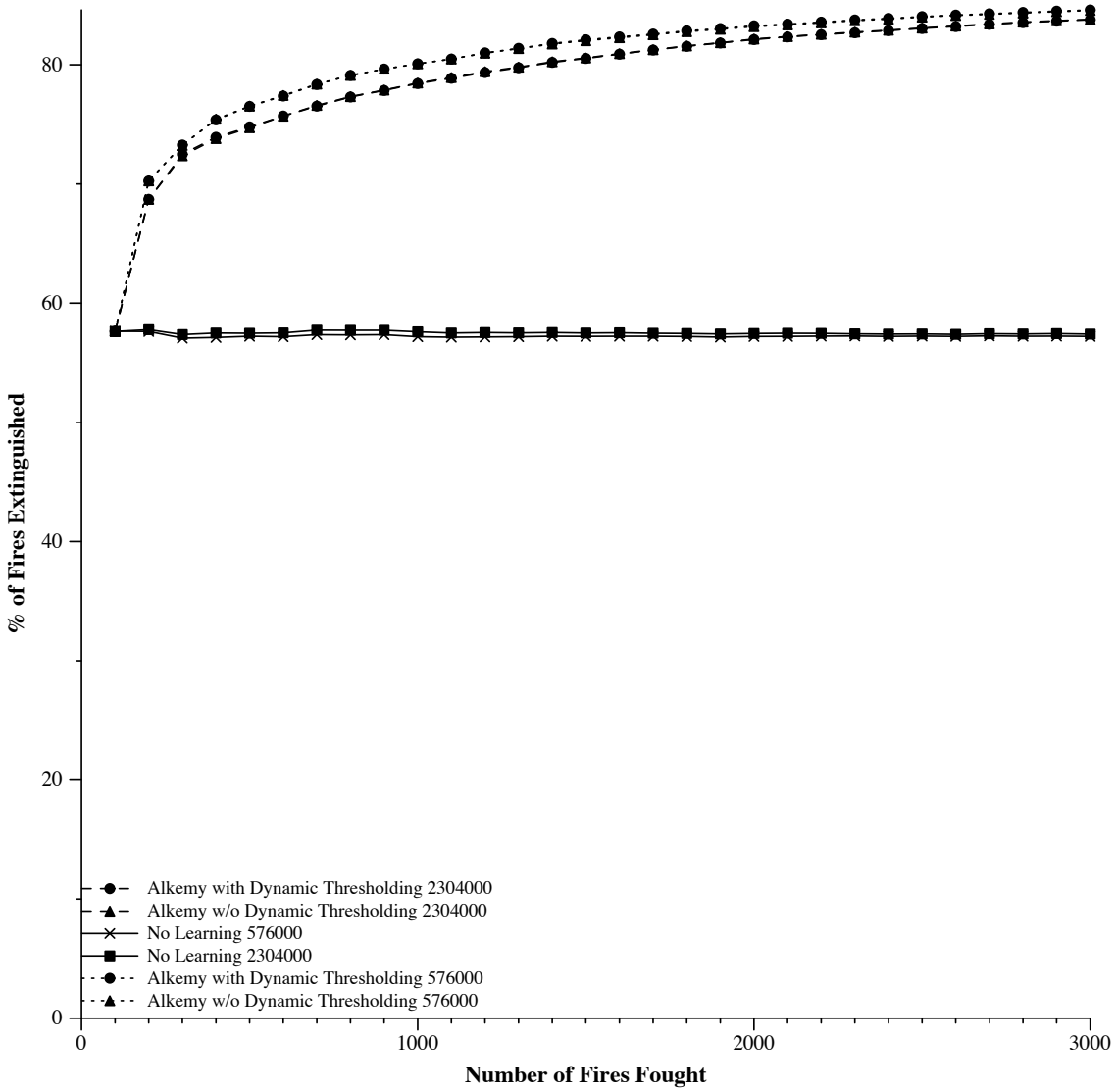

Figure 4.3: Experimental results for Alkemy with and without Dynamic Thresholding

thresholding makes no difference to the accuracy of the Alkemy learner.

The reason why Alkemy was not affected by dynamic thresholding was because the tree returned by Alkemy was already highly accurate. Regardless of the threshold which was set dynamically, Alkemy always returned an accuracy that was greater than or equal to the threshold which therefore rendered the adjustments made by dynamic thresholding useless. It also appears that a static threshold of 0.5 is enough for the agent to assess the accuracy of the Alkemy decision tree. As seen in Figure 4.4, the accuracy of the decision tree returned by Alkemy rises very quickly and remains at roughly 85%.

Overall, based on the results in Figure 4.3 it can be concluded that dynamic thresholding would only be at its most effective when used in conjunction with learners that are less capable of producing highly accurate results, when the results of a learner can not be trusted and therefore need to be monitored and more stringently tested or in dynamic domains where rules for determining success change and therefore the need to trust the learner when new situations arise becomes important.

4.3.2 The Simile Algorithm And Its Effects on Statistical Learning

Research goal (3) addresses the issue of whether analogous reasoning (the Simile algorithm) can be useful in boosting the accuracy of the Statistical learner by referring to similar past cases. Our hypothesis for this research goal was that analogical reasoning, when used in conjunction with certain kinds of learning, can be a means of improving an agent's performance. The more complex a domain, the more possibilities exist within that domain. Hence, as the domain size increases the chance of a past case being 'useful', in the sense that prior knowledge contains identical situations and hence a possible answer, becomes increasingly smaller. For example, if a domain contains only 36 possible states, then the chances of finding a past case that is identical is very likely. However, if the size of the search space is increased to 2,304,000 different possibilities, the chances decrease significantly. The Simile algorithm allows the agent to exploit knowledge of *similar* cases which are used in conjunction with identical past cases.

With regards to the Simile algorithm, the process of deciding if a given state S_1 is 'easier' than another state S_2 is done via a function $Simile(S_1, S_2)$ which compares all variables in state tuple S_1 to those in state tuple S_2 and returns whether S_1 is 'easier' or 'harder' than S_2. The numeric values which accompany each value in the input file are used to determine difficulty, the higher the value meaning the more difficult. For example, given the variable

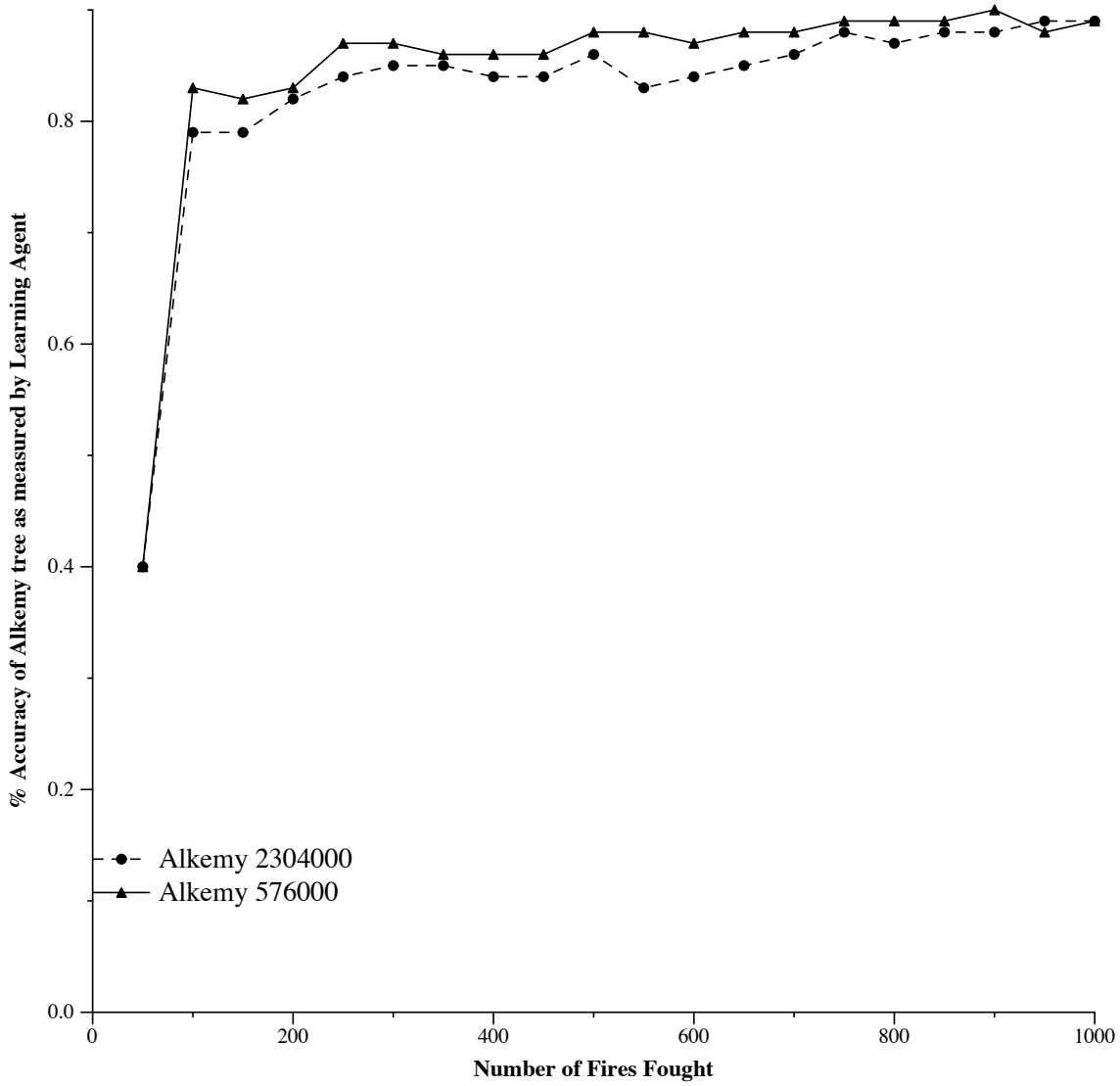

Figure 4.4: Average accuracy of tree produced by Alkemy

FloorType. 'Ceramic' is easier than 'Wood' as Wood has a difficulty of 3 compared to Ceramic which has 0.

The way in which we conducted our experiments in relation to the Simile algorithm involved the activation of a boolean parameter 'Simile On' within our JACK learning agent system. We ran a set of 50 experiments, each with 3000 fires for both Simile active and inactive in both domain sizes.

As seen in Figure 4.2, it is clear that the Simile algorithm combined with Statistical reasoning performs significantly better than just Statistical reasoning alone. It can also be concluded that larger, more complex domains require more careful reasoning to derive more accurate predictions. When we look at Figure 4.2, the most dramatic change with respect to the Simile algorithm occurs with the addition of exceptions to the rules. When the problem domain is made more difficult, the accuracy of Statistical learning without Simile is reduced to that of no learning. Therefore it can be said that analogical reasoning greatly increases the accuracy of Statistical learning especially in non-linear domains where exceptions in the rule-set exist and hence is one mechanism that a learning agent may use to process its past experiences and helps improve performance. While not enough to make it as accurate as Alkemy, Simile does prove to be useful as opposed to no learning. It is interesting to see that not even the Simile algorithm can prevent the plateauing of the Statistical method at about 600 fires with exceptions added to the domain. T-test results from the comparison of Simile against no Simile also confirm this and we can confidently say that the Simile algorithm makes a statistically significant difference[3].

4.3.3 Statistical Clusters and their effects on learning

In relation to research goal (4), the aim was to investigate whether the placing of thresholds on Statistical Learning improved the accuracy of the agent. In order to achieve this goal, we introduced a 'Statistical cluster' into the original Statistical learning algorithm. The use of *Clustering* in Statistical learning can be seen as an additional form of restriction on the result-set returned by the original Statistical learning algorithm in that a further subset of the full set returned by the learner is created. This subset is known as the **cluster**. The purpose of incorporating a cluster into the Statistical learning algorithm is to 'broaden' the number of possible solutions by increasing the number of retardants that are recommended by the Statistical learner. By restricting the solution to a single retardant with the highest

[3]Domain size 2304000 p-value < 0.0005; Domain size 576000 p-value < 0.0005

probability of success, the agent runs the risk of ignoring retardants that fall just slightly below the accuracy of the highest scoring retardant. These ignored retardants may potentially prove useful in that the agent may apply further reasoning on a **set** of retardants rather than a **single** retardant. Acceptance into the cluster was determined by a *clustering threshold*. The cluster acceptance threshold represents the difference between the retardant with the highest statistical chance of success and another retardant (which may or may not have the same chance of success). If the difference is less than or equal to a pre-defined threshold, then the retardant being compared is accepted into the cluster.

In order to experiment with these types of thresholds we ran a set of experiments which use *Clustering* to assist the agent in determining whether to follow a learner's recommendation. During experimentation with the Clustering Statistical Algorithm, we varied the threshold of acceptance into the cluster and observed the effects this had on the size of the cluster. This in turn affected the effectiveness of the learning agent as too high a threshold would allow every retardant into the cluster. If that were the case, then the cluster would be a very poor discriminator in selecting a truly useful retardant, especially in the case of the more difficult fires where a reduced number of retardants are able to extinguish the fire. We also experimented with static thresholds, that is having a given numeric threshold that did not change throughout that experiment. This was given to the agent via a command line parameter at the beginning of each experiment. The values used can be seen in Table 4.5. The way in which the experiments for clustering were conducted were that we altered the threshold of acceptance of a retardant into the cluster and observed the resulting accuracy. These threshold values were chosen arbitrarily and range from 1.0 to 0.0007. For our analysis of the results we will be comparing clustering with non-clustering. Clustering was not used with Alkemy as it always returns a single solution regardless of the situation, hence no chance is given to cluster any other retardants.

As seen in Figure 4.5 the effect of not using clustering with Statistical learning is a significant drop in accuracy of approximately 15.5% in the smaller domain and 18.7% for the larger domain after 300 fires. However, this gap is reduced after 3000 fires with only a 2.14% difference existing in the smaller domain and 6.38% in the larger domain. The results in Figure 4.5 demonstrate that clustering is effective overall when compared to no clustering however its benefits are limited. This can be seen in the steady rise in accuracy of the non-clustering algorithm while at the same time, the clustering algorithm does not improve much after 800 fires. In the more complex domain, clustering is very beneficial to the agent. The reasons for the large dip between the first 100 and 1000 fires was because of

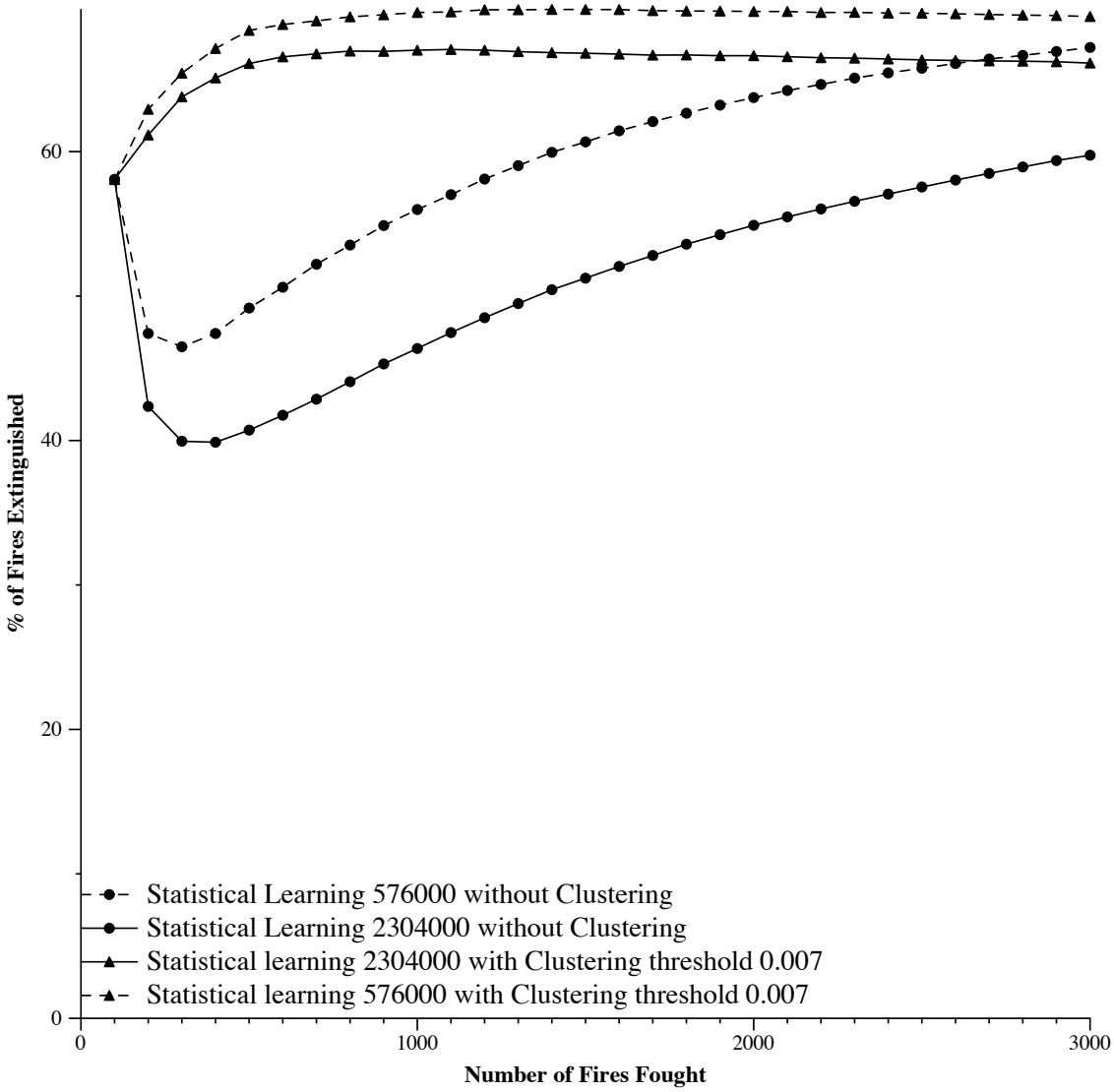

Figure 4.5: Experimental results for Statistical learning without Clustering

the small threshold used, 0.007 and the random choice of retardants within the cluster. The small threshold of 0.007 meant that only retardants which were most likely to succeed were selected. In addition to this, the random selection of retardants from the cluster meant that a larger variety of retardants was available to the agent which in turn increases the chances of finding one that extinguishes the fire successfully. A T-test analysis of the results in Figure 4.5 against no learning shows that only the larger domain size is statistically significant with a p-value of less than 0.0005 while the smaller domain size yielded a p-value of 0.081. While not within the 0.05 limit of acceptance it is still reasonable to say that Statistical without clustering in the smaller domain size does make some difference. When comparing the results of Statistical learning with clustering against Statistical learning without clustering we found that these results are statistically significant with a p-value of less than 0.0005.

Threshold	Average Accuracy Of Predictions(%)	Average Cluster Size
0.0007	72.87	2.38
0.006	65	3.8
0.007	66.00	3.82
0.0074	64.27	3.97
0.009	63.63	4
0.0148	60.7	4.47
0.04	58	4.85
1.0	56.86	5

Table 4.2: Thresholds and their effect on cluster sizes

Algorithm	Time taken in large domain	Time taken in small domain
Statistical with Clustering	60m 23.12s	60m 55.18s
Statistical without Clustering	60m 38.8s	61m 9.69s

Table 4.3: Times taken for Clustering Statistical

As seen in Table 4.2 there is a clear relationship between the value of the cluster acceptance threshold, the size of the cluster and the accuracy of the agent's predictions. As the threshold value increases, so do the number of retardants that are accepted into the cluster and at the same time the accuracy of the agent decreases. The highest accuracy is gained when the threshold value is set to 0.0007 resulting in an average cluster size of 2.38 and an

average accuracy of 72.87%. This means that when the difference between the retardant with the highest chance of success and the other remaining retardants was less than or equal to 0.0007, 72.87% of the 3000 fires it fought were successfully extinguished. When the threshold was increased to 0.0074 we attained an average cluster size of 3.97 and an accuracy of 64.27%. The lowest accuracy is attained when the threshold is set to 1.0, resulting in a cluster size of 5 and an accuracy of 56.86%. This accuracy is roughly equivalent to no learning at all.

In Table 4.3 the times taken for clustering were approximately the same for both domain sizes. When compared to no clustering, the clustering algorithm is approximately one to two minutes slower over a set of 50 runs of 3000 fires. In general, not much of a difference is made in terms of time however the accuracy of the Statistical learner with clustering active tends to be higher than no clustering at all.

It appears that as the cluster size increases, the accuracy of the agent decreases. The reason for this is that as the threshold becomes smaller, fewer retardants are likely to satisfy that smaller threshold and hence the more exclusive the cluster becomes. This leads to fewer retardants being added to the cluster therefore the more effective they are likely to be, hence the accuracy of predictions rises. In a sense, the agent is becoming more selective in its choice of retardants and as a result, only the 'best' retardants will be selected. In general, it is expected that the lower the threshold value, the more accurate the predictions become. Overall, the use of clustering is effective in increasing the accuracy of the Statistical learner. The lowest threshold used of 0.0007 produced an accuracy of 72.87% while without clustering the accuracy was approximately 66.13%.

4.4 Pruning Historical Cases For Efficiency

The goal of research question (5) was to address the issue of pruning historical cases to see what effects this had on the performance of the agent. In answering the question of what effect pruning the agent's historical case-set would have on the performance and accuracy of the agent, we implemented the *Sliding Window* algorithm. This algorithm is given a single numeric value which acts as a window size by which to restrict the history set that is accessed by the learner. The Sliding Window algorithm can be seen as the opposite of the Simile algorithm where past cases are taken away or 'pruned' instead of being added. Intuitively, it would appear that restricting the number of cases that are given to a learner would lead to an increase in performance in terms of *time* and *resources*. However, it is not entirely clear what effect this would have on the *accuracy* of the learner. Our hypothesis is

that for domains that change frequently, the omission of old and possibly irrelevant cases will lead to an increased accuracy. For static domains which do not change, the accuracy of the learner would depend highly upon the size of the window. This is because if the size of the window is too small, not enough cases would be given to the learner and hence a lower accuracy would result. If the window size is too large, a higher accuracy may result however the learner would be processing more cases than it would need to achieve this.

We designed a series of experiments that utilise the *Sliding Window* algorithm to test whether restricting historical cases to pass into the learner can increase the efficiency of the agent as well as examine what effects this would have on the accuracy. The values given represented the maximum number of historical cases to pass into the learner. The window sizes we chose were 200, 300 and 500. For example, if a value of 300 was chosen as the window size, the learner (either Alkemy or the Statistical learner) would be given *at most* 300 historical cases to learn from.

When we compare Figure 4.2 where no Sliding Window is used with Figures 4.6, 4.7 and 4.8 it is interesting to see that Alkemy is the most effective learning algorithm, despite a drop in accuracy of approximately 10% when using Sliding Window for all sizes. Statistical with Simile also loses accuracy with a 3% drop when using Sliding Window for all window sizes. Statistical learning without Simile is the least affected with virtually no difference at all for all window sizes.

When altering the size of the window on Alkemy, the trend for both search space sizes was that as the size of the window increased, the accuracy of the learner also increased, although only slightly. As the size of the window increases from 200 to 500, the accuracy of the learner increases in accuracy by roughly 1%.

With regards to Figures 4.6, 4.7 and 4.8, some of the T-test results were statistically significant when compared against their non-Sliding Window counter-parts. For example, Alkemy with Sliding Window compared with Alkemy without Sliding Window showed that all window size results for both domain sizes were below the threshold of 0.05[4]. However, the Statistical learning results were not all statistically significant when compared against no Sliding Window. With regards to Statistical learning with Simile, only window size 200 produced statistically significant results for both domain sizes[5] while window sizes 300 and 500 produced p-values between 0.33 and 0.46 in the smaller domain size. Most notably, Statistical learning with Simile in the large domain with Sliding Window 300 had a p-value

[4] p-value < 0.0005

[5] 2304000 p-value $= 0.002$; 576000 p-value $= 0.001$

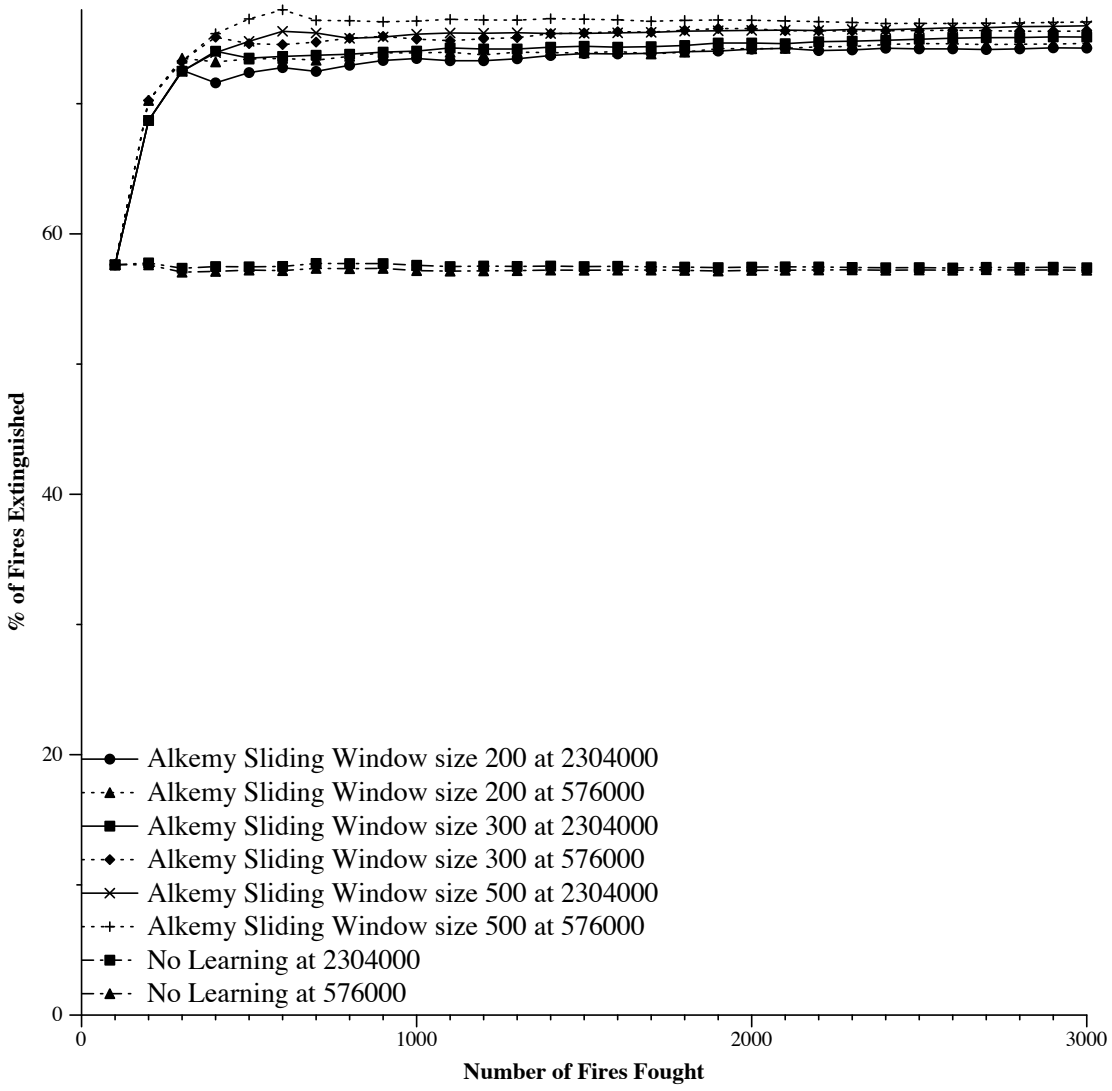

Figure 4.6: Sliding Window results for Alkemy

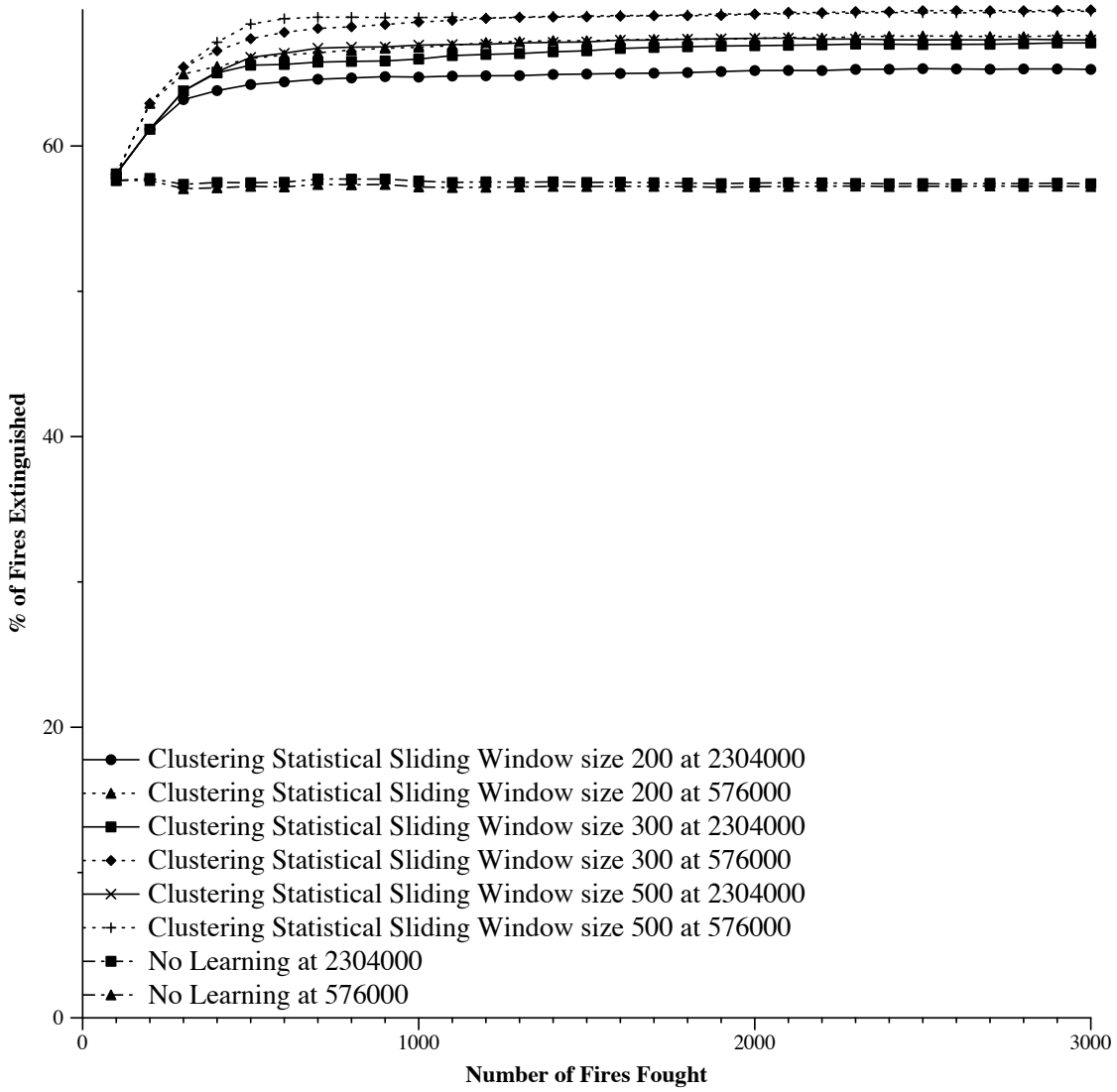

Figure 4.7: Sliding Window results for Statistical with Simile

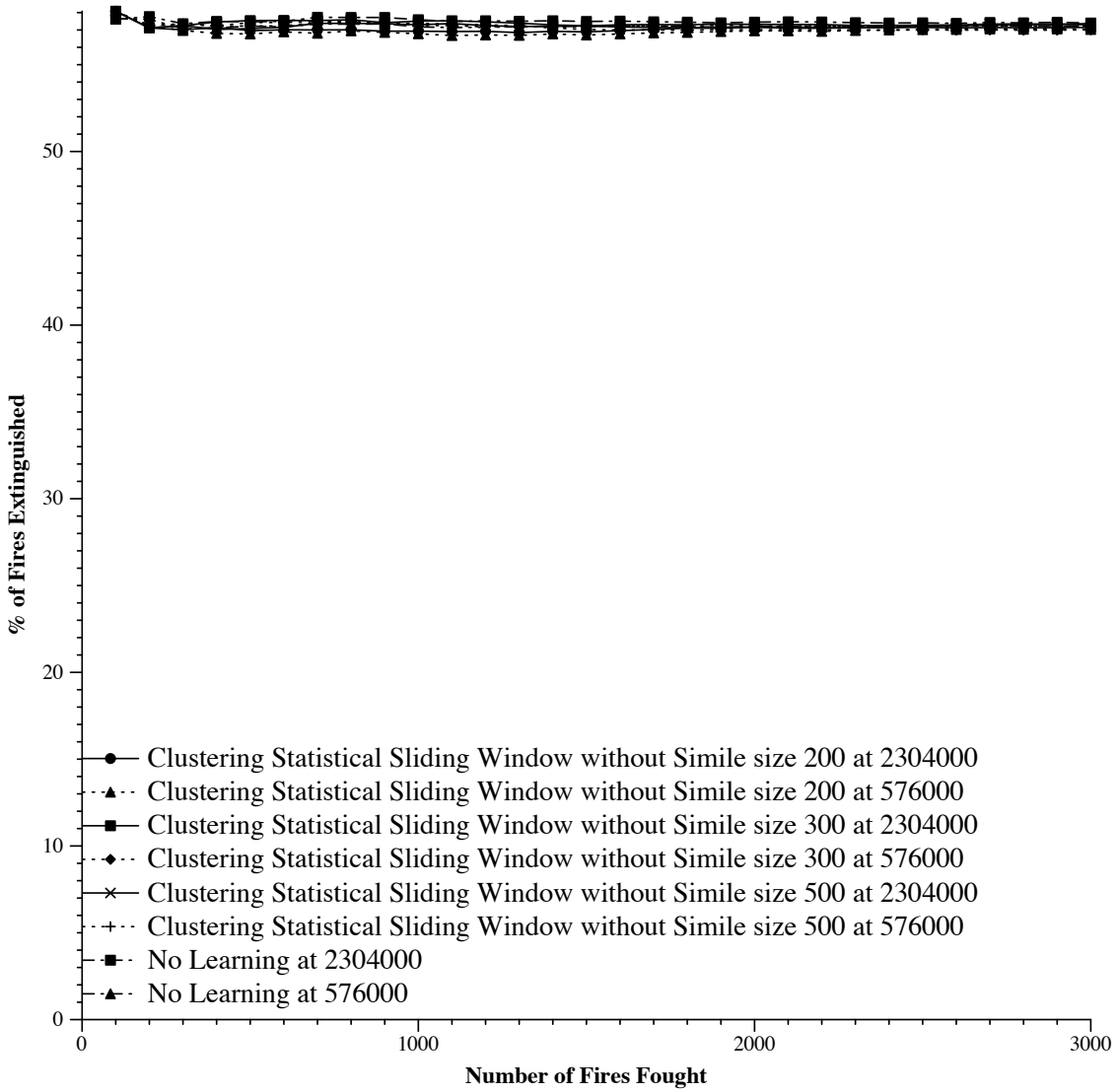

Figure 4.8: Sliding Window results for Statistical without Simile

of 0.930 while in the same domain, window size 500 yielded a p-value of 0.390. Looking at Statistical learning without Simile the smaller domain size had more statistically significant results than the larger one with window sizes 200[6] and 500[7] having p-values less than 0.05 in the smaller domain. In comparison, the larger domain size only had one statistically significant result of 0.047 at window size 500. Window size 300 produced a p-value of 0.071. A very close p-value of 0.057 for window size 200 however, shows that the Sliding Window algorithm was able to make some difference to Statistical learning without Simile but not enough to allow us to say that it is a significant difference.

Algorithm	Time taken in large domain	Time taken in small domain
No Learning	1m 36.12s	1m 36.26s
Alkemy	6h 17m 21.65s	5h 46m 52.34s
Statistical with Simile	5m 17.39s	5m 17.67s
Statistical without Simile	5m 25.65s	5m 23.62s

Table 4.4: Times taken for Sliding Window at size 200

Algorithm	Time taken in large domain	Time taken in small domain
No Learning	1m 35.7s	1m 37.75s
Alkemy	6h 45m 23.88s	6h 9m 56.19s
Statistical with Simile	6m 50.23s	6m 53.74s
Statistical without Simile	7m 12.92s	7m 17.86s

Table 4.5: Times taken for Sliding Window at size 300

Algorithm	Time taken in large domain	Time taken in small domain
No Learning	1m 36.82s	1m 36.31s
Alkemy	9h 3m 25.69s	8h 7m 36.25s
Statistical with Simile	9m 53.89s	10m 4.05s
Statistical without Simile	10m 16.31s	9m 56.32s

Table 4.6: Times taken for Sliding Window at size 500

[6] < 0.0005
[7] 0.026

As seen in Tables 4.4, 4.5 and 4.6 the timings for the Sliding Window experiments show that as the window size increases, so does the time taken to produce a recommendation. Alkemy still takes the longest of the learning algorithms yet the accuracies of Alkemy using Sliding Window show that it still out-performs the Statistical learner. The main differences to note are that as the size of the Sliding Window increases from 200 to 500, the differences in time between the two domain sizes becomes larger for Alkemy than with Statistical learning. With window size 200, the difference in times between Alkemy at search space size 2,304,000 and 576,000 is 8% compared to the Statistical learners at 1%. When the window size is increased to 300 fires the difference is 8% for Alkemy while the Statistical learners is still 1%. When the window size is 500 fires the time difference becomes 10% for Alkemy while Statistical with Simile increases to 2% and Statistical without Simile becomes 4%.

Overall, it appears that restricting the amount of historical cases that are given to the learner reduces the accuracy of the learner. In particular, learners that are highly dependant upon domain knowledge, such as Alkemy, are more affected than relatively simpler numerical learners such as Statistical Learning.

4.5 When Should The Agent Learn?

With regard to research question (6), our goal is to explore how frequently the agent should learn. We will vary the number of fires an agent experiences from 100, 200, 300 and 500 before giving those experiences to the learning module. Our hypothesis is that when the agent learns too often the benefit of learning is off-set by the resources used such as disk space and memory. Conversely, if the agent does not learn often enough it would save in computational resources however any performance benefit from learning would not be fully utilised. Hence by altering the frequency of how often the agent learns we hope to discover what effect this will have on the accuracy of predictions. This alteration in the frequency of learning only affects Alkemy and not Statistical learning because unlike Alkemy, Statistical learning does not produce a result that can be used across multiple fire scenarios. While Alkemy creates a decision tree that can be referenced on numerous occasions, Statistical learning *must* be run for every fire as the recommendation given is based upon a statistical tally of retardants used in all fires encountered.

We have devised a series of experiments whereby the frequency of learning is altered. During experimentation, the agent monitors how many fires it has fought and once it reaches a certain interval, the learner is activated and learning output is produced and used. This

frequency in learning is *static*. By 'static' we mean that it is not changed during the course of the experiment. The values we chose for the agent to learn were every **100**, **200**, **300** and **500** fires. As an example, if the learning frequency value was set to 200, the agent would learn every 200 fires meaning it would first activate learning at 200 fires followed by learning at 400 fires then 600 etc... If learning frequency value was set to 500 the agent would first learn at 500 fires followed by learning at 1000 fires then 1500 fires etc... An alternative approach to changing how *often* the agent should learn is to monitor the accuracy of the predictions and to apply learning only when the accuracy begins to drop below a certain threshold, however due to time constraints this particular technique was not implemented.

In Figure 4.9 we see that as the frequency of learning increased, the accuracy also increased indicating that the more the agent knows about its environment, the more discernment there is between varying states it encounters. In contrast, if the agent knows fewer rules about its world, then it is able to categorise states less effectively, leading to a greater variation in accuracy as it is less able to determine whether a retardant will work or not. An interesting feature from Figure 4.9 is that after 3000 fires, the accuracies become closer to each other and when considering time constraints, it seems plausible to learn at shorter intervals at first at the cost of time, but then move to longer intervals such as every 500 fires after fighting 3000 fires. This saves time while the accuracy stays relatively the same. Hence, from the results in Figure 4.9 we can conclude that the agent should learn frequently at first, and when the accuracy reaches a 'saturation point' where it does not change or changes only slightly, the agent should learn *less frequently*. Although the results in Figure 4.9 were obtained from a static environment, the same principles apply in dynamic environments except that the agent would have to monitor changes in its predictive accuracy more often and after some number of learning periods without any significant change in accuracy, the agent should learn less frequently.

When comparing the various frequencies of learning against the original frequency of every 100 fires, the T-test analysis showed that all results except Alkemy at frequency 200 in the larger domain size[8] were statistically significant, meaning that the performance of the agent was affected by altering the frequency of learning in those cases. The T-test results for Alkemy in the smaller domain were all statistically significant[9].

With regard to the times taken for the frequency of learning experiments to complete, Alkemy took between approximately 12.2 hours to 37.78 hours to fight 3000 fires depending

[8]Frequency 200 = 0.110; Frequency 300 = 0.005; Frequency 500 < 0.0005

[9]Frequency 200 = 0.009; Frequency 300 = 0.001; Frequency 500 < 0.0005

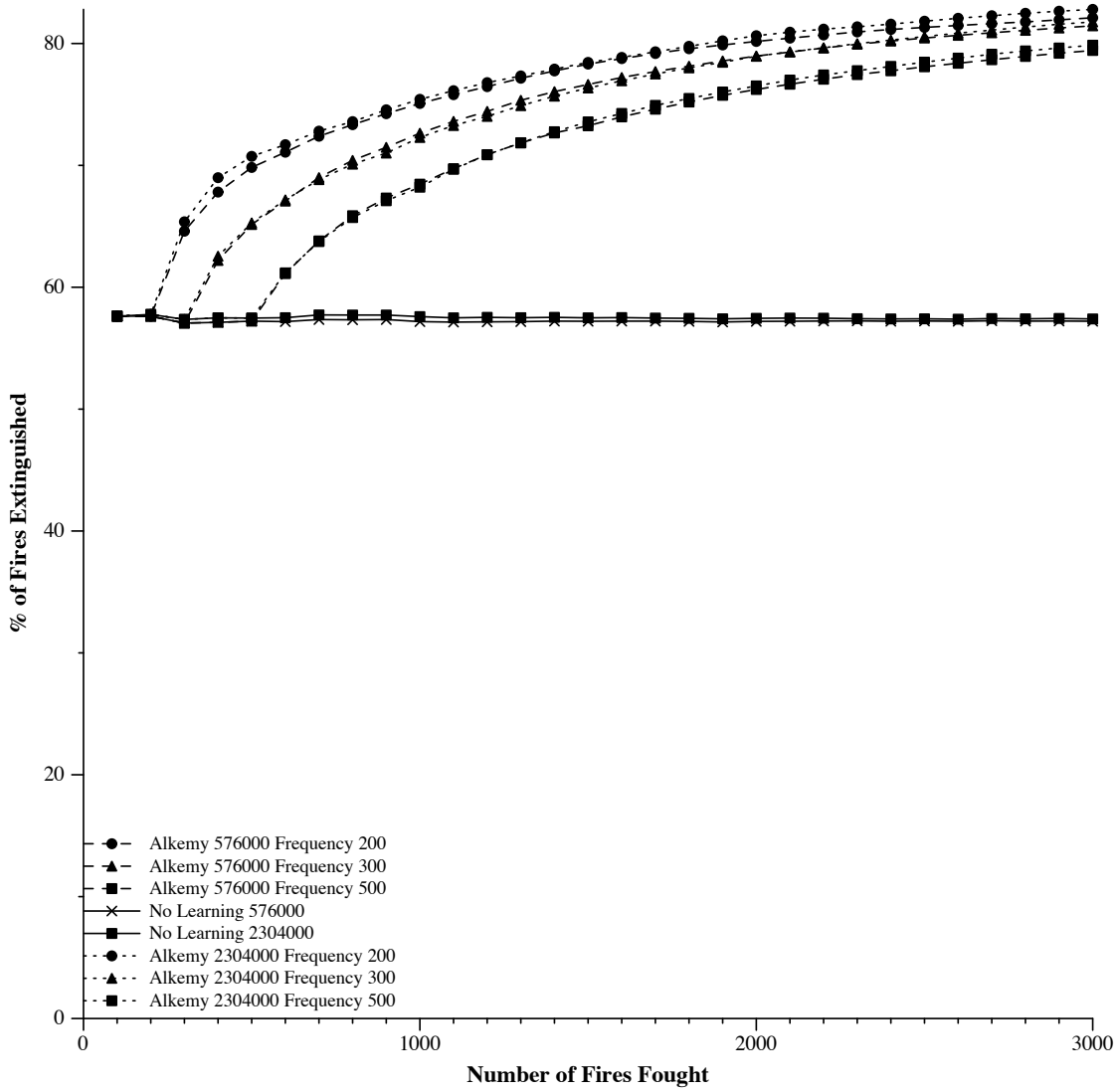

Figure 4.9: Frequency of Learning for Alkemy

on how frequently the agent learnt. As the frequency of learning decreased for Alkemy, so did the time in a roughly inversely proportionate manner. For example, if the agent was only learning every 300 fires, it took roughly 1/3 of time it normally takes if it learnt every 100 fires in the large domain while in the small domain it took roughly 1/2 of the time. For learning frequency 500, it took roughly 1/4 of the normal time in the large domain and 1/3 of the time in the smaller domain.

-	Search Size 576,000	Search Size 2,304,000
Learning Frequency 100	53h 58m 12s	44h 55m 52.77s
Learning Frequency 200	25h 43m 57.26s	22h 54m 12.53s
Learning Frequency 300	18h 12m 13.24s	16h 40m 50.05s
Learning Frequency 500	12h 22m 13.64s	12h 11m 42.39s

Table 4.7: Time taken for Alkemy for various learning frequencies

4.6 Domain Characteristics And Their Effects On Learning

With the introduction of exceptions to the rule-set, we begin to see greater differences between the learning algorithms as seen in Figure 4.2. In these new conditions, Simile makes a significant difference to the point where without it, the agent performs just as accurately as no learning. In addition to this, Alkemy appears to out-perform Statistical learning in terms of accuracy. So by introducing exceptions to the problem, a very different set of results is achieved which demonstrates that depending upon the specific domain the agent is learning under, one of either Inductive or Statistical learning should be used and that learning is a domain specific problem.

Comparing the different search space sizes, as expected, the statistical method's performance degrades as the search space size increases. However, Alkemy's performance doesn't generally appear to be significantly affected by the search space and in fact Alkemy does slightly better in terms of % of fires extinguished when the search space is larger. This implies that inductive learning is better able to learn given a more complex domain than statistical learning.

As seen in Figure 4.10, the only statistically significant results are the comparisons of Statistical with Simile and no learning. This allows us to conclude that Statistical learning with analogous reasoning and no learning are affected by the changes in search space size.

- No Learning at 2,304,000 Vs No Learning 576,000: p-value < 0.0005
- Alkemy at 2,304,000 Vs Alkemy at 576,000: p-value $= 0.375$
- Statistical with Simile 2,304,000 Vs Statistical with Simile at 576,000: p-value < 0.0005
- Statistical without Simile 2,304,000 Vs Statistical without Simile at 576,000: p-value $= 0.449$

Figure 4.10: T-test results of search space size comparison in complex domain

Alkemy's p-value of 0.375 indicates that the change in search space size can not allow us to confidently concluded that Alkemy's results are significant. Therefore we can conclude that Alkemy is not as sensitive to domain size changes as Statistical learning. Statistical without Simile has a high p-value which indicates that no significant difference occurs when the domain size changes.

It is interesting to see that in Figure 4.2 there is little difference in the results across the two search space sizes for Alkemy in Figure 4.9. This reinforces the fact that Alkemy is insensitive to search space size change. With respect to Figure 4.6 there is only a 1% difference in the accuracy of Alkemy which is in favour of the smaller search space. This suggests that Alkemy is not affected by the four-fold increase in search space size as the standard deviation for both the upper and lower bounds are 4%-3% respectively.

With respect to Sliding Window, all p-values of the comparison between search space size using Sliding Window for Alkemy are above the threshold of 0.05[10]. This indicates that the Alkemy set of experiments is not statistically significant enough to conclude that a difference exists when the search space size is increased from 576,000 to 2,304,000. What this does indicate however is that Alkemy is less sensitive to domain size changes than Statistical learning with Simile[11] and no learning[12]. With respect to Statistical learning without Simile, only window size 500 was statistically insignificant with a p-value of 0.562. Window sizes 200 and 300 for Statistical without Simile were 0.008 and 0.050 respectively.

[10]Window size 200 = 0.571; Window size 300 = 0.307; Window size 500 = 0.354
[11]Domain size 200 < 0.0005; Domain size 300 < 0.0005; Domain size 500 = 0.002
[12]p-value < 0.0005

4.7 General Discussion

Clearly, learning is beneficial to the agent's performance. As seen in Figures 4.1 and 4.2, the smaller search space (576,000) produces slightly more accurate results than the larger search space. No learning produces the least accurate predictions with exceptions providing a slightly greater challenge by reducing the chance of random success by approximately 2%.

The general trend for all the learning algorithms is that at the early stages of the experiments, the accuracy of the learners rises very sharply and by about 400 fires, begins to stabilise within the low 80% accuracy range. Within the larger domain of size 2,304,000, Simile does approximately 1% better than without Simile for the first 1000 fires. This difference increases to 4% after 3000 fires. The reason why this is the case is because of the inverse relationship between size and the chances of encountering the exact same state again. The odds of finding the exact same state in a search size of 36 is

$$\text{Search Space Size } 36 = \frac{1}{36}$$

as opposed to a less likely chance of

$$\text{Search Space Size } 2{,}304{,}000 = \frac{1}{2,304,000}$$

Hence the larger a search space gets, the chances of finding an exact match become smaller.

In using Simile, we have relaxed the constraint that exact matching imposes. Instead, Simile asks for historical cases that are exact *as well as* those that match to a certain degree of similarity. This increases chances of finding a larger set to reason with. Yet, as the search space decreases, the chances of finding a subset within a certain degree of similarity become greater since there are fewer combinations. This reduced number of combinations increases the chances of finding more cases that are similar to a given scenario. This would explain why Statistical with Simile in the smaller search space (576,000) performs better than Statistical with Simile in the larger search space (2,304,000).

Another interesting trend can be seen in the plateau that occurs for Statistical learning with Simile. After 3000 fires, the curve for Statistical learning with Simile is near horizontal whereby no further significant difference can be seen. Alkemy appears to still be improving after 3000 fires. Overall, learning in the smaller search space yields a 10% improvement over no learning with a 27% improvement in the larger search space.

As seen in Figure 4.2, Alkemy and our Statistical learning algorithm still perform better than no learning at all. Yet, a greater difference between the learning algorithms exists where

Alkemy clearly produces more accurate predictions for this data set. The smaller search space of 576,000 combinations produces more accurate predictions than the larger search space. However, the main difference this time compared to Figure 4.1 is that the Simile algorithm with Statistical reasoning performs significantly better than Statistical reasoning alone. In fact, the Statistical method without Simile performs equally with that of no learning. In addition to this, Alkemy is clearly a more accurate learner for this data set than the Statistical algorithm.

Chapter 5

Related Work

There have been various approaches to learning in agents. The largest body of work in agent learning stems from the machine learning area, where learning techniques such as Bayesian learning, reinforcement learning, Q-learning and neural networks appear to be the most popular approaches to learning [Kudenko et al., 2003]. Evolutionary computation approaches have used genetic algorithms and genetic programming in order to evolve new behaviours [Meyer, 1997]. Logical approaches that take advantage of the declarative semantics that are available, use Inductive Logic Programming (ILP) which allows the use of background knowledge to help generate more general hypotheses [Kudenko et al., 2003]. Reasoning techniques such as case-based reasoning (CBR) and explanation-based learning (EBL) have been used as a means of trying to emulate human thought processes where past experiences are recalled if similar scenarios are encountered [Kudenko et al., 2003]. Multi-agent approaches try to make greater use of individual agent knowledge and expertise through the sharing of experiences and hypotheses [Kudenko et al., 2003; Tan, 1997]. This has proven effective in some domains yet at the cost of communication overhead and additional space costs. Biological approaches use Darwinian and Lamarckian evolution and introduce the notion of evolvable languages as well as agent behaviours and capabilities [Kudenko et al., 2003]. From a psychological perspective, using *recognition primed decision making* as a form of reasoning enables agents to detect 'cues' which may indicate certain scenarios so that an agent is able to recognise similar instances of past experience [Norling, 2001]. Anthropology has been used in the context of agent migration from one 'society' to another [Bordini and Campbell, 1995]. In this approach, each society has an agent that is responsible for making sure that new agents are provided with the knowledge required to function effectively in that society.

More specifically, in relation to the work presented in this thesis, the body of work that encompasses the extension of BDI-like agent systems with learning capabilities has been less extensive than in the multi-agent domain. As far as the author is aware, no other single agent system combines inductive learning, statistical learning, analogical reasoning and dynamic thresholding together as seen in this thesis. Some researchers in the field have extended BDI systems with inductive learning while others have extended BDI systems with case-based reasoning [Olivia et al., 1999]. All these different approaches have some commonality to our work and will be presented in this Chapter.

Work into extending BDI agent systems, such as JACK can be seen in the work of Sioutis and Ichalkaranje [2005] with the Cognitive Hybrid Reasoning Intelligent Agent System (CHRIS). What Sioutis et al. propose is that instead of using Bratman's BDI model alone, they combine two other human decision making models: (1) Rasmussen's Decision Ladder and (2) Boyd's OODA Loop. The Observe, Orient, Decide and Act (OODA) loop is a model of decision making created by Hammond [2004]. It describes that the act of deriving an appropriate action consists of four primary stages from the initial events that enter the system (observation), the interpretation of that information based upon such factors as past experience and cultural heritage (orientation), the decision that is then made (decision) and finally the action that results as a culmination of the previous three factors (Act). Rasmussen's Decision Ladder is a methodology that formalises the stages of the OODA loop [Rasmussen et al., 1994]. The type of learning that *CHRIS* uses is traditional reinforcement learning. CHRIS separates its learning into two components: *active* learning and *passive* learning. Active learning is the execution of reinforcement learning. Passive learning is when the reinforcement learning's action-selection policy is replaced with a JACK plan. By doing this, the agent's plan set is gradually altered to include learning behaviour. In essence, this allows the agent to reflect upon its own past behaviours. In comparison, the work presented in [Sioutis and Ichalkaranje, 2005] is similar to our model in that it is also not 100% generic in the sense that the specifics of the domain must be encoded by the agent designer. Both our system and that of [Sioutis and Ichalkaranje, 2005] can then be customised based on the specific domain. The main difference between our model is that we use inductive and statistical learning combined with analogical reasoning (Simile), dynamic thresholding and the sliding window algorithm.

The extension of plans through learning can also be seen in the work of Sevay and Tsatsoulis [2002] where a set of plans are run with examples repeatedly to adapt to diverse scenarios. Case-based reasoning is used as the adaptation mechanism in [Sevay and Tsatsoulis, 2002] while our work uses inductive and statistical learning with dynamic thresholding

and the sliding window algorithm.

Subagdja and Sonenberg [2005] propose a Q-Learning extension to the BDI framework through the use of meta-level plans. Three primary stages are used to allow a BDI agent to adapt through learning: (1) Generating a hypothesis by executing a meta-plan (2) Testing the hypothesis and (3) Constructing or altering plans based on successful hypotheses. Subagdja et al regard learning as a series of pre-defined plans that are executed which result in adaptive behaviour. Design patterns are used in their model to constrain the search whilst constructing such meta-level learning plans. The difference with our work is that we use inductive and statistical learning combined with analogical reasoning, dynamic thresholding and sliding windows for historical case exclusion.

The PRODIGY project [Veloso et al., 1995] is an architecture that concentrates on improving the quality of plans generated and the efficiency of the planner (plan generator). However, although PRODIGY uses many different learning algorithms, such as inductive logic programming, to improve certain aspects of plan generation, it is not primarily centered around the BDI architecture. Also off-line learning algorithms are not used by PRODIGY. What we propose is to be able to exploit times where the agent may be idle in certain domains and to use this time to perform learning or reasoning to improve the performance of the agent. In addition to this, the idea of dynamically adjusting the trust threshold of learnt data is not proposed in the PRODIGY architecture.

The work of Nguyen and Wobcke [2006] also focuses on learning within the BDI architecture whereby the Alkemy learner [Ng, 2004] is used. Their domain focusses on a Smart Personal Assistant (SPA), which helps users to remotely manage and manipulate their e-mail and calender. The SPA agent accepts verbal commands from the user and learns their preferences over time. For example, if an e-mail is particularly large, the SPA agent will only display a summary of the e-mail. Although the domains are different, the similarities between our work and that of Nguyen and Wobcke include the use of the JACK Intelligent Agents Toolkit and Alkemy. The differences are that the do not study the effects of how varying the frequency of learning and dynamic thresholds affect the learning accuracy of the agent. Nguyen et al. do not investigate dynamic thresholding, do not use statistical learning as part of their experimentation, nor do they explore the area of analogous reasoning as we have with the Simile algorithm. The system proposed by Alonso and Kudenko [2000] also uses Inductive Logic Programming through a combination of Explanation Based Learning (EBL) and ILP. EBL uses only one past case to generalise a rule while our statistical method considers all past cases. The main difference is our model uses Simile to adjust the agent's

decision bias based on previously similar cases as well the use of dynamic thresholding.

The work of Cole et al. [2003] also adds learning to agents. Their primary goal was to design a symbolic machine learning system which could accurately derive Q functions and policies. Being able to describe the domain through a descriptive hypothesis language was also one of their main issues. Their problem is based upon the blocks world domain and learning is achieved through the Alkemy learner. They have similar assumptions to our work in that they assume the agent has some knowledge of the application domain. Their learning model is also similar to ours in that they have a 'policy library', a central reasoning engine and the learning component. The main differences between this work and ours are that they have no similarity measures as we have with our Simile algorithm, and they have no notion of dynamic thresholding. In our work we varied how often the agent learnt however this remained static throughout the experiments while the work of Cole et al. [2003] only invoked the Alkemy learner every 100 cases. They also use a sliding window of past cases where a window size of 200 was used for their Q-function search tree and a window size of 1000 was used for their policy tree. Similar work is also conducted by Cole et al. [2005] where Alkemy is used as the learner for a TV show recommender. High levels of personalisation were achieved with the TV recommender yielding an average accuracy of 84.69% compared to that of AdaBoost with 85.34%. The main differences between this work and ours is that the TV recommender is multi-agent while ours is a single agent. Also, their work does not have any similarity measures or dynamic thresholding. With regards to past cases being ignored or deleted, the work of Cole et al. [2005] prompt the user to make sure the most irrelevant cases are ignored. That is, if any inconsistencies in data arise then the TV recommender will ask the user for feedback as to what the correct value should be. Our sliding window algorithm does not prompt the user and deletes the oldest cases.

As our work is centered around goal-plan agents with an interest in the BDI framework, the role that goals play are of interest. The work done on this topic by Leake and Ram [1995] shows that there are different types of goals with respect to learning, in particular, learning can be *goal-driven* or *goal-relevant*. The difference is that goal-relevance is not explicitly dictated by the reasoner, yet produces output that is still relevant to the agent's goals. Goal-driven learning is directly related to what the agent's goals are. Using the terminology of [Leake and Ram, 1995], it can be said that our learning agent was goal-driven towards putting out fires with the fire data being the goal-relevant information. However, the work of Ram et al. does not mention the idea of similarity and the dynamic adjustment of trust thresholds.

The notion of *similarity* when referring to past cases has also been applied to reinforce-

ment learning [Ribeiro et al., 2002]. Ribeiro et al. describe a 'spreading' effect that a new case has on past cases where given a degree of similarity, a reinforcement is *spread* to more than one action. Hence, a cascading effect is created. The main difference is that the approach developed by Ribeiro et al. causes a permanent change to the learning data which is in turn used by the learner. Since reinforcements are applied to other related states, the reinforcement for all those states is adjusted either positively or negatively with the encompassing experience. Our approach does not alter stored cases, rather, for each unique case it retrieves a subset of the agent's experiences that are related to it. So if the current environmental situation were to change, a potentially different subset of cases would be retrieved. This process does not change the cases being retrieved, but simply changes the nature of the query. In this way, any negative, potentially non-reversible changes are avoided. In addition to this, Ribeiro et al. focus on multi-agent reinforcement learning while we focus on single agent inductive and statistical learning.

The concept of reducing the amount of data to compute in order to lessen the overhead associated with learning has been a primary research question in this thesis and has been realised through our Sliding Window algorithm. However, the notion of a *Sliding Window* is also used as a means of reducing the complexity of search spaces in the area of reinforcement learning [Ono and Fukumoto, 1996]. Here, Ono et al. only scan portions of the total search space and divide it into smaller portions. However, the work of Enembreck and Barthes [2005] suggests that pruning and similarity measures are not needed to allow an agent to learn efficiently. They propose *Entropy-based* learning where instead of using traditional methods of learning such as reinforcement learning or inductive learning, the data is represented as a *concept graph*. Incoming data is incrementally added to the graph which is then used to guide the agent's actions. The work done by Foner and Maes [1994] describes *what* an agent should learn with a small emphasis on *when* it should learn. However it does mention off-line learning as a type of dream state the agent may enter in which time it should reflect upon what information it has gathered and to try to learn from that data. Their work shows that selective filtering of certain information can be used to reduce the complexity of the tasks the agent must achieve. This is similar to our Sliding Window and Simile approach however the idea of dynamically adjusting the trust threshold of learnt data is not present in [Foner and Maes, 1994].

In relation to when an agent should learn, Joshi [1996] presents a learning system based on Bayesian belief nets, neural networks and fuzzy logic to probabilistically determine solutions. The main point of their work is that an agent should learn when problems occur. An agent

should not learn when it is too computationally expensive and should instead use alternate methods of deriving a solution, i.e modelling. Another solution to the "*When should an agent learn?*" problem can be seen in the work of Schmidhuber and Zhao [1996] where the combination of *Evaluation points* and past experience are used to set learning intervals. Evaluation points are stages where the agent thinks about its past actions. When an agent reaches an evaluation point it begins to learn, hence the agent sets way points for periodic history evaluation.

SOAR [Laird et al., 1987], a rule based agent system that uses a learning technique known as *chunking* to create plans. Chunking is executed whenever *impasses* occur. An impass is when an agent cannot solve a problem. Our model is different in that we learn new information regardless of problems occurring, which allows for exploratory behaviour.

The work done by Lynden and Rana [2002] attempts similar work to ours in the sense that they extend an agent toolkit (FIPA-OS) to include learning, which was not part of the original design of the toolkit. Specifically, they combine reinforcement learning and neural networks into a learning module for FIPA-OS agents to use. However, the main difference is that the FIPA-OS toolkit is not modelled using the BDI architecture.

The work of Muggleton et al. [1999] introduces *close loop machine learning* to completely automate the process of agent action selection and execution. Close loop machine learning is where the agent selects actions as well as carries out those actions autonomously, whereas active learning only selects actions. The JACK agents we used in our work already had the capability of selecting actions and executing them. Integrating Alkemy and statistical learning provided the JACK agent a means by which to learn in much the same way as active learning was used in Bryant's work. However, the notion of similarity and the use of a Sliding Window of experience is absent in [Muggleton et al., 1999].

Although the notion of agents that modify themselves is not new as shown by Brazier and Wijngaards [2001], the mechanism that facilitates this feature is *external* to the agent, through the use of *agent factories*. These factories provide the elemental building blocks and templates that are put together to create a whole agent. What we propose is to have the adaptation mechanism *within* the agent. Also, their approach does not use any learning algorithms as such while our approach does.

The work presented in [Buffet et al., 2002] describes scalable adaptivity whereby behaviours which have already been learnt are recombined as a way of incrementally altering the agent's behaviour. The way in which this is done is by combining two or more learnt rules to create a more generic rule that covers more cases. In our approach, the learnt knowledge

returned by the learning module may contain two or more rules however they are queried as separate rules. Our work does not recombine learnt knowledge but instead queries the same knowledge differently.

The views that adaptability can be generic in that different learning modules can be added or removed to adjust performance can be seen in [Selfridge and Feurzeig, 2002]. Selfridge et al. present a component-based view of agent adaptation with the introduction of *Elementary Adaptive Modules* (EAM's). Our work also takes a modular approach to learning in that the learning module may be extended to use any type of learning algorithm. The difference with our work is that we do not change modules as in [Selfridge and Feurzeig, 2002], instead we have the one module that queries its knowledge base in a different way, depending on the current situation.

The notion of deleting past cases in order to improve the time taken for learning has been explored by Smyth and Keane [1995]. Their work focussed on the removal of historical cases in order to reduce the amount of space used to store information as well as preserving the accuracy of the learner. In comparison, our work was done within the BDI Agent framework while the work of Smyth et al. was done within the machine learning domain. Our work does not involve the deletion of cases, instead we use such techniques as similarity matching which affects a different subset of the historical case set and the use of a 'sliding window' where cases after a certain time frame are omitted.

Chapter 6

Conclusion

We have presented a model that introduces learning into the BDI framework. This model allows beliefs to be generalised through inductive learning and statistical tallying. We have developed and experimentally tested, various analogous reasoning algorithms which use contextual and relative reasoning to alter agent behaviour according to past experience. In general, we have experimentally shown that learning can be a useful tool which allows an agent to improve its performance by a considerable amount.

Overall, the most accurate of the learning mechanisms we explored is the Alkemy inductive learner with the statistical learner being the second most accurate while no learning at all produces the least accurate results. It was interesting to see from the graphs that statistical learning plateaus while Alkemy does not. This indicates that given more domain knowledge through training data, Alkemy can potentially produce more accurate results. This shows a limitation in the predictive accuracy of statistical learning and how symbolic learning can produce greater accuracy than simpler numeric learning. However, this greater accuracy comes at a time cost ranging from 4 to 54 times longer than statistical learning. Another interesting pattern we discovered experimentally was that Alkemy is less sensitive to changes in the search space size than statistical learning. When altering between 576000 and 2,304,000 different possible fire state combinations, the results produced by Alkemy were very similar. Statistical learning on the other hand was clearly affected by the change in complexity of the domain. This implies that inductive learning is better able to describe a given domain than statistical learning.

The use of the Simile algorithm improves the performance of Statistical learning by approximately 12%. Without using Simile reasoning, the agent performs as well as no learning

at all. As expected, the Simile algorithm performs approximately 4% better in the smaller search size than the larger search size. This was because the chances of coming across a similar or related case would be more likely if less possibilities exist. These results indicate that Statistical learning, coupled with analogical reasoning produce more accurate results than just Statistical learning alone. In a sense, analogical reasoning serves to 'boost' the accuracy of Statistical learning by 'widening' the number of cases being considered before an answer is derived. However, as seen in the experimental results we can also conclude that the Simile algorithm and dynamic thresholding serve to boost only 'weak' learners, which produce low accuracies as they do not take into account the majority of the environmental state to compute a learnt prediction. If a learner is able to richly describe a domain such as with higher order functions and be able to create and search through various hypotheses from a given hypothesis space as Alkemy does, then it would be able to produce fairly high accuracies without the aid of additional algorithms. Hence these 'strong' learners would not require any boosting.

We have discovered that by using a 'Sliding Window' approach to restricting the amount of history that is feed into the learner, the accuracy of the results was reduced by 3%-10% although a benefit was that this took much less time than passing in all historical cases. The primary result of these experiments was that inductive learning, particularly Alkemy, is more sensitive to changes in the training sample size than statistical learning as Alkemy drops in accuracy by 10% while statistical learning with Simile only drops 3%. Alkemy still remains the most accurate learner while benefiting from a 80%-86% decrease in time taken to produce a recommendation. Hence, the Sliding Window algorithm should be deployed when agents use inductive learning and not when they use statistical learning.

We have experimentally shown that by decreasing the threshold of acceptance into a statistical cluster of potential retardant choices, the agent is able to increase its predictive accuracy. By decreasing the threshold of acceptance, we are making it more difficult to be added into the cluster of potential choices hence only the most effective retardants are finally chosen from which results in a more accurate outcome.

As a final note, we would like to acknowledge that these results were obtained for a given domain, with given data. A key issue is how to generalise to other domains: will Alkemy still outperform Statistical learning for other domains or other data? Further experimentation will be required to answer such a question.

6.1 Future Work

A potential extension to our work could include using XML as a means of making our model more generic. At present, the agent designer must know prior to execution of the system, the format of the learnt data. XML can be used to help agent designers to more easily add any new learning algorithms to our system. The description must still be written, however, writing this description in a more abstract and conceptual way would most certainly aid in the usability of our system. In a sense, this approach can be considered as a way of introducing *meta-learning* descriptors into agents.

Another avenue of interest is the issue of *when* to learn. At present, it is only set to a static frequency is used. The problem is that using a static variable to tell an agent when to learn is highly domain dependant. Unless the optimal frequency of learning is known prior to running the system, over time the system will degrade in one of two ways. If the frequency of learning is set too high, the agent will perform unnecessary learning which would use up resources and may potentially slow down the agent. If the frequency is set too low, the learning agent will under-perform as it is not taking advantage of knowledge stored in its history.

Ideally, learning should be done every time a new event arrives to ensure that the best solution is given. However, this can be expensive. A potential solution could be to use the Simile algorithm to see how much of Unseen is related to all of Seen or a subset of Seen. If most of the new cases, say 90%, in Unseen are not related to the subset of Seen, then they are new cases that haven't been seen before and so Alkemy should be called to learn. But if most (90%) of the cases in Unseen are related/similar to Seen, then no learning is needed since the existing tree covers most of these cases.

The basis of the Simile algorithm was to measure, within a certain degree, the similarity of one case to another. Therefore the primary purpose of analogous reasoning was to find other similar past cases. Another application of the Simile algorithm could be as a deletion mechanism. As illustrated in this work, the more fires the fire fighter agent fights, the more cases are stored. This increase in past instances leads to a slowdown, which in the case of Alkemy, can be very dramatic. Hence, deleting cases would be beneficial as it would save computational resources. The way in which deletion of cases could occur is by using Simile to classify cases that are similar to ones already seen by the learning mechanism. All cases that are similar to ones already seen could be deleted. The cases that are *not* similar could be kept as input for the next round of learning. The rationale behind this is to keep cases

you haven't seen before and to delete those that you have or are similar to ones you have seen. This approach would be an extension of the *Sliding Window* algorithm in the sense that rather than not considering a subset of the total history, we would be *deleting* the history instead.

An extension of the *Sliding Window* mechanism for selecting only a subset of the total history recorded by the agent can be further extended by changing what the 'window' is composed of. Instead of selecting a set of historical cases based on the one fixed sized window, the window can consist of non-contiguous cases. This allows a more precise selection of historical cases that may provide more insight than the last 'x' cases. This effectively fragments the window and moves it away from being a static structure.

At present, the Simile algorithm refers to all variables in the fire state. If all variables are 'easier' or 'harder' then those cases will be considered for use otherwise they are classified as 'not useful'. An extension of this notion can be a further relaxation of this constraint where every variable in the state must be 'easier' or 'harder'. Instead, we can have *partial* state matching to cover even more states rather than classifying them as useless. How much is considered may even vary from state to state. It may be set to 90% similarity for successful classification or to 80% etc... However, this approach has a limitation which assumes that there are no inter-relationships between any of the variables of the state.

As indicated there are a range of possible directions for future work. This thesis has provided a starting point by which a generic framework of learning for BDI agents has been introduced to help alter their behaviour based on past experience. We have explored and experimented with Inductive and Statistical learning, analogous reasoning, historical pruning of past cases, dynamic trust adjustment, frequency of learning and domain characteristics which affect learning.

Appendix

The timing experiments were carried out on a personal computer with the following specifications:

Processor:	Pentium P4 2.8GHz
RAM:	512MB (DDR)
Hard disk drive:	Seagate ST360021A ATA (120GB, 2MB Cache, 7200RPM)
Operating system:	Fedora Core 2.0

Bibliography

E. Alonso and D. Kudenko. Logic-Based Multi-Agent Systems for Conflict Simulations. Technical Report ILCLI-00-FCSAI-4, The University of York, York, UK, 2000.

E. Alonso and D. Kudenko. Machine Learning Techniques for Adaptive Logic-based Multi-Agent Systems. In *Proceedings of The Second Workshop of the UK Special Interest Group on Multi-Agent Systems*. Springer-Verlag, 1999. URL citeseer.nj.nec.com/alonso99machine.html.

R. A. Belecheanu, S. Munroe, M. Luck, and T. Payne. Commercial Applications of Agents: Lessons, Experiences and Challenges. In *Proceedings of the Fifth International Joint conference on Autonomous Agents and Multiagent Systems*, Hakodate, Hokkaido, Japan, May 2006. Springer-Verlag; Berlin Germany.

R. Bordini and J. Campbell. Towards An Anthropological Approach To Agent Adaptation. In *Proceedings of the First International Workshop on Decentralized Intelligent and Multi-Agent Systems (DIMAS'95)*, 1995.

M. E. Bratman. *Intentions, Plans, And Practical Reason*. Harvard University Press, Cambridge, MA, first edition, 1987.

F. Brazier and N. Wijngaards. Automated Servicing Of Agents. *AISB Journal: Special Issue on Agent Technology*, 1(1):5–20, 2001.

J. Bruske, I. Ahrns, and G. Sommer. Practicing Q-Learning. In *Proceedings of the 4th European Symposium on Artificial Neural Networks (ESANN06)*, 1996.

O. Buffet, A. Dutech, and F. Charpillet. Learning To Weigh Basic Behavious In Scable Agents. In *Proceedings of the First International Joint Conference on Autonomous Agents*

and Multiagent Systems, pages 1264–1265, Bologna, Italy, July 2002. ACM Press; New York, NY, USA.

C. Carabelea. Adaptive Agents in argumentation-based negotiation. In *Multi-Agent-Systems And Applications*, pages 180–187, 2001. URL citeseer.nj.nec.com/492049.html.

P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artif. Intell.*, 42(2-3): 213–261, 1990. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(90)90055-5.

J. Cole, J. Lloyd, and K. Ng. Symbolic Learning For Adaptive Agents. In *Proceedings of the Annual Partner Conference*, pages 139–148. Smart Internet Technology Cooperative Research Centre, September 2003.

J. Cole, M.Gray, J. Lloyd, and K. Ng. Personalisation for User Agents. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multi Agent Systems, AAMAS-05*, pages 603–610. Springer-Verlag: Berlin, Germany, July 2005.

D. E. Comer. *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture*. Prentice Hall, first edition, 1995.

S. L. de Medeiros Rivero, B. H. Storb, and R. S. Wazlawick. Economic Theory, Anticipatory Systems And Artificial Adaptive Agents. In *Proceedings of the 4th Conference on Information Systems Analysis and Synthesis - ISAS'98*, pages 64–69, 1998. URL citeseer.nj.nec.com/110125.html.

M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification Of dMARS. In *Agent Theories, Architectures, and Languages*, pages 155–176, 1997. URL citeseer.ist.psu.edu/dinverno97formal.html.

S. Dzeroski, L. D. Raedt, and H. Blockeel. Relational Reinforcement Learning. In *International Workshop on Inductive Logic Programming*, pages 11–22, 1998. URL citeseer.nj.nec.com/dzeroski98relational.html.

F. Enembreck and J.-P. Barthes. ELA: A New Approach For Learning Agents. *Autonomous Agents and Multi-Agent Systems*, 10(3):215–248, 2005. ISSN 1387-2532. doi: http://dx.doi.org/10.1007/s10458-004-6976-8.

L. Foner and P. Maes. Paying Attention To What's Important: Using Focus Of Attention To Improve Unsupervised Learning. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior '94, MIT Press, Brighton, UK*, 1994.

M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The Belief-Desire-Intention Model Of Agency. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10. Springer-Verlag: Heidelberg, Germany, 1999. URL citeseer.ist.psu.edu/georgeff99beliefdesireintention.html.

M. P. Georgeff and A. L. Lansky. Reactive Reasoning And Planning. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, USA, July 1987.

G. Hammond. *The Mind At War: John Boyd and American Security*. Smithsonian Institution Press, first edition, 2004.

J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, first edition, 2001.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, first edition, 2001.

P. J. Hoen, V. Robu, and H. L. Poutre. Decommitment In A Competitive Multi-Agent Transportation Setting. In R. Unland, M. Klusch, and M. Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 409–431. Whitestein Series in Software Agent Technologies, Birkhauser Publishing, Springer-Verlag Group, 2005.

M. J. Huber. JAM: A BDI-Theoretic Mobile Agent Architecture. In *Proceedings of the 3rd International Conference on Autonomous Agents*, pages 236–243, Seattle, USA, 1999.

T. Iba and Y. Takefuji. Adaptation Of Neural Agent In Dynamic Enviroment: Hybrid System Of Genetic Algorithm And Neural Network. In *Proceedings of the 2nd International Conference on Knowledge-Based Intelligent Electronic Systems*, 1998.

N. R. Jennings. An Agent-Based Approach For Building Complex Software Systems. *Communications of the ACM*, 44(4):35–41, 2001.

A. Joshi. To Learn Or Not To Learn. In G. Weiss and S. Sen, editors, *Adaptation and Learning in Multi–Agent Systems, IJCAI'95 Workshop*, pages 127–139. LNCS Springer-Verlag: Heidelberg, Germany, 1996. URL citeseer.ist.psu.edu/joshi96to.html.

D. Kudenko, D. Kazakov, and E. Alonso. *Machine Learning For Agents And Multi-Agent Systems*, chapter 1, pages 1–26. Idea Group Publishing: Hershey, PA, first edition, 2003. ISBN 1-59140-046-5.

J. Laird, A. Newell, and P. Rosenbloom. SOAR: An Architecture for General Intelligence. *Artificial Intelligence*, 3:1–64, 1987.

D. Leake and A. Ram. *Chapter 1: Learning, Goal And Learning Goals*, pages 1–43. MIT Press/Bradford Books, Cambridge, MA, first edition, 1995.

H. Lee, P. Mihailescu, and J. Shepherdson. *Teamworker: An Agent-Based Support System For Mobile Task Execution*, chapter 7, pages 433–448. Software Agent-Based Applications, Platforms and Development Kits. Birkhauser-Verlag, 2005.

A. Lin and J. Debenham. Incorporating Adaptive Interaction Property In Cooperative Agent For Emergent Process Management. In *Proceedings of the Third International Symposium on Cooperative Database Systems for Advanced Applications(CODAS)*, pages 110–117, 2001.

S. Loke. An Overview Of Mobile Agents In Distributed Applications: Possibilities For Future Enterprise Systems. *Informatica: An International Journal of Computing and Informatics*, 25(2):247–260, July 2001.

S. Lynden and O. F. Rana. LEAF: A FIPA Compliant Software Toolkit For Learning Based MAS. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1260–1261, Bologna, Italy, July 2002. ACM Press; New York, NY, USA.

T. Matsui, N. Inuzuka, and H. Seki. A Proposal For Inductive Learning Agent Using First-Order Logic. In *J. Cussens and A. Frisch, editors, Work-in-Progress Reports of ILP-2000, pages 180-193, London, UK*, 2000.

J.-A. Meyer. Biomimetic Mechanisms Of Self-Organisation And Adaptation In Animats. In F. Hara and K. Yoshida, editors, *Proceedings of the International Symposium on System Life*, 1997.

M. Mitchell. *Machine Learning*. McGraw Hill, New York, first edition, 1996.

P. A. Mitkas, D. Kehagias, A. L. Symeonidis, and I. N. Athanasiadis. A Framework For Constructing Multi-Agent Applications And Training Intelligent Agents. In *The Fourth International Workshop on Agent-oriented Software Engineering (AOSE 2003) held at Autonomous Agents and Multi-Agent Systems (AAMAS 2003)*, Melbourne, Australia, 14-18 July 2003.

S. Muggleton. *Inductive Logic Programming*. Academic Press, first edition, 1992.

S. Muggleton, C. Bryant, C. Page, and M. Sternberg. Combining Active Learning With Inductive Logic Programming To Close The Loop In Machine Learning. In S. Colton, editor, *Proceedings of the AISB'99 Symposium on AI and Scientic Creativity*, 1999. URL citeseer.ist.psu.edu/bryant99combining.html.

K. Ng. Alkemy: A Learning System Based on an Expressive Knowledge Representation. Available from http://users.rsise.anu.edu.au/~kee/Alkemy/, August 2004.

A. Nguyen and W. Wobcke. An Adaptive Plan-Based Dialogue Agent: Integrating Learning Into a bdi Architecture. In *Proceedings of the Fifth International Joint conference on Autonomous Agents and Multiagent Systems*, Hakodate, Hokkaido, Japan, May 2006. Springer-Verlag.

E. Norling. Learning To Notice: Adaptive Models Of Human Operators. In D. Precup and P. Stone, editors, *Agents-2001 Workshop on Learning Agents, Montreal, Canada*, 2001.

J. Odell. Objects and Agents Compared. *Journal of Object Technology*, 1:41–53, 2002.

C. Olivia, C. Chang, C. Enguix, and A. Ghose. Case-Based BDI Agents: An Effective Approach for Intelligent Search on the World Wide Web. In *AAAI Spring Symposium*, 1999.

N. Ono and K. Fukumoto. A Modular Approach To Multi-Agent Reinforcement Learning. *distributed ARTIFICIAL INTELLIGENCE MEETS MACHINE LEARNING: Learning in multi-agent environments*, 1221:25–40, 1996.

F. B. Pereira and E. Costa. The Influence of Learning in the Behavior of Information Retrieval Adaptive Agents. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 452–457. ACM Press New York, NY, USA, 2000.

A. S. Rao and M. P. Georgeff. BDI-Agents: From Theory To Practice. In *Proceedings of the First International Conference on Multiagent Systems*, 1995.

J. Rasmussen, A. Pejtersen, and L. Goodstein. *Cognitive Systems Engineering.* Wiley & Sons, first edition, 1994.

C. H. C. Ribeiro, R. Pegoraro, and A. H. R. Costa. Experience Generalization for Concurrent Reinforcement Learners: the Minimax-QS Algorithm. In *Proceedings of the 1st International Conference on Autonomous Agents and Multi-Agent Systems*, 2002.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, second edition, 2003.

J. A. Sauter, R. Matthews, H. V. D. Parunak, and S. Brueckner. Evolving Adaptive Pheromone Path Planning Mechanisms. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 434–440, Bologna, Italy, July 2002. ACM Press; New York, NY, USA.

J. Schmidhuber and J. Zhao. Multi-Agent Learning With The Success-Story Algorithm. *distributed ARTIFICIAL INTELLIGENCE MEETS MACHINE LEARNING: Learning in multi-agent environments*, 1221:82–94, 1996.

O. G. Selfridge and W. Feurzeig. Learning In Traffic Control: Adaptive Processes And EAMs. In *Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN '02)*, pages 2598 –2603, 2002.

H. Sevay and C. Tsatsoulis. Multiagent Reactive Plan Application Learning In Dynamic Environments. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 839–840, Bologna, Italy, July 2002. ACM Press; New York, NY, USA.

D. Singleton. An Evolvable Approach To The Maes Action Selection Mechanism. Master's thesis, The University Sussex, Brighton, UK, 2002. URL citeseer.nj.nec.com/536400.html.

C. Sioutis and N. Ichalkaranje. Cognitive Hybrid Reasoning Intelligent Agent System. In *Proceedings of the 9th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems (KES '05)*, pages 838–843, Melbourne, Australia, September 2005. Springer-Verlag; Berlin Germany.

R. Smith and N. Taylor. A Framework For Evolutionary Computation In Agent-Based Systems. In *Proceedings of the 1998 International Conference on Intelligent Systems*, pages 221–224, 1998. URL http://www.ics.uwe.ac.uk/~rsmith/fecabs.pdf.

R. Smith, P. Kearney, and W. Merlat. Evolutionary Adaptation In Autonomous Agent Systems - A Paradigm For The Emerging Enterprise. *BT Techol J*, 17(4), 1999.

B. Smyth and M. Keane. Remembering To Forget: A Competence Preserving Case Deletion Policy For CBR Systems. In *Proceedings of the International Joint Conferences on Artifical Intelligence(IJCAI'95)*, pages 377–382, Montreal, Canada, 1995.

B. Subagdja and L. Sonenberg. Learning Plans With Patterns Of Actions In Bounded-Rational Agents. In *Proceedings of the 9th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems (KES '05)*, pages 30–36, Melbourne, Australia, September 2005. Springer-Verlag; Berlin Germany.

A. L. Symeonidis, P. A. Mitkas, and D. D. Kechagias. Mining Patterns And Rules For Improving Agent Intelligence Through An Integrated Multi-Agent Platform. In *Proceedings of the 6th International Conference on Artificial Intelligence and Soft Computing(ASC2002)*, Banff, Alberta, Canada, 17-19 July 2002.

M. Tambe, D. Pynadath, N. Chauvat, A. Das, and G. Kaminka. Adaptive Agent Integration Architectures For Heterogeneous Team Members. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS'00)*, pages 301–308, Los Alamitos, CA, USA, 2000. IEEE Computer Society. URL citeseer.nj.nec.com/tambe00adaptive.html.

M. Tan. Multi-Agent Reinforcement Learning: Independent vs. Cooperative Learning. In M. N. Huhns and M. P. Singh, editors, *Reading In Agents*, pages 487–494. Morgan Kaufmann, San Francisco, CA, USA, first edition, 1997.

M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, and J. Blythe. Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 1995.

D. Weyns, A. Helleboogh, and T. Holvoet. *The Packet-World: A Test Bed For Investigating Situated Multi-Agent Systems*, chapter 7, pages 383–408. Software Agent-Based Applications, Platforms and Development Kits. Birkhauser-Verlag, 2005.

M. Wooldridge. *An Introduction To MultiAgent Systems.* John Wiley and Sons Ltd, first edition, 2002.

T. Zhang and S. Covaci. Adaptive Behaviours Of Intelligent Agents Based On Neural Semantic Knowledge. In *Proceedings of the 2002 Symposium on Applications and the Internet(SAINT'02)*, 2002.